CHAPTER 3
ONE-MACHINE CLASS SCHEDULING PROBLEMS

In Chapters 3, 4, and 5, we discuss the results of our research into a number of different scheduling problems and the general job shop scheduling problem. The primary motivation is to examine subproblems of the job shop scheduling problems in order to gain insight into the larger problem. The subproblems that we will examine are interesting scheduling problems that have not been previously considered.

We will start with the one-machine class scheduling problems. Our approach in this chapter is to develop analytical results, to test extended heuristics, and to show that a problem space genetic algorithm can be a good procedure for a variety of scheduling problems.

3.1 Introduction

The three one-machine problems studied in this work are class scheduling problems that model the complicating factor of machine setups in the manufacturing process. Most problems with sequence-dependent setup times are NP-complete. Class scheduling problems have a special structure that makes them good candidates for further research: the jobs to be scheduled form a number of disjoint job classes and setups occur whenever the machine processes consecutive jobs from different classes. And although we will study a number of different problems, we will see that a problem space genetic algorithm will be a useful procedure for all of them.

The one-machine class scheduling problems under investigation are as follows:

1. Constrained Flowtime with Setups (CFTS)
2. Class Scheduling with Release and Due Dates (CSRDD)
3. Flowtime with Setups and Release Dates (FTSRD)

The first problem studied is the class scheduling extension of the one-machine problem of minimizing the total flowtime of a set of jobs that have deadlines. The research includes an optimality property for the jobs in the same class, a good heuristic, and the development of a problem space genetic algorithm that can find better solutions.

The second problem is the class scheduling problem where each job has a release date and a due date. The objective is to minimize the number of tardy jobs. We investigate a number of heuristics and the use of a problem space genetic algorithm. We also look at a secondary criteria, minimizing the total tardiness.

The objective in the third problem is to minimize the total flowtime where each job has a release date. We consider a number of approaches to finding good solutions, including a problem space genetic algorithm, and present a special case that can be solved with a pseudo-polynomial dynamic programming algorithm.

This chapter considers the research on each of these problems in turn. Research relevant to these problems is also discussed in Chapter 2. See especially Sections 2.6 and 2.7.

3.2 Constrained Flowtime with Setups

We will first consider the one-machine class scheduling problem of minimizing the total flowtime subject to the constraint that each job must finish before its deadline. A new heuristic is proposed for the problem. We investigate the use of a genetic algorithm to improve solution quality by adjusting the inputs of the heuristic. We present experimental results that show that the use of such a search can be a successful technique.

3.2.1 Introduction

This research is motivated by the scheduling of semiconductor test operations. Assembled semiconductor devices must undergo electrical testing on machines that can test a number of different types of semiconductors. If a machine is scheduled to test a lot consisting of devices that are different from the devices tested in the previous lot, various setup tasks are required.

These tasks may include changing a handler and load board that can process only certain types of semiconductor packages or loading a new test program for the new part. However, if the new lot consists of circuits that are the same as the previous type, none of this setup is required. This type of change is a sequence-dependent setup that can be modelled by class scheduling.

Since post-assembly testing is the last stage in semiconductor manufacturing, meeting a job's due date is a very important objective for the manager of a test facility. A secondary criterion is the minimization of total flowtime (the sum of the job completion times), which reflects the manager's desire to increase throughput and decrease inventory holding costs.

This is a dual criteria problem, in which the primary criterion is used as a constraint and the secondary criterion is optimized under this restriction. The problem of minimizing the total flowtime subject to deadlines is an old problem. Smith (1956) provides an optimal solution technique that repeatedly schedules the longest eligible job last. The class scheduling version of the problem, however, is more difficult.

For our problem, finding a feasible schedule is an NP-complete problem. (A schedule is feasible if every job finishes before or at its deadline.) Thus, there exist no exact algorithms to minimize in polynomial time the total flowtime subject to the deadline constraints. (For a discussion of the theory of NP-completeness, see Section 2.10 and Garey and Johnson, 1979.) Thus, we are motivated to try different heuristics. In this work we develop a multiple-pass heuristic that finds good solutions quickly. The first contribution of our investigation of this problem is the extension of Smith's algorithm into a heuristic which considers the setup times while sequencing the jobs by their deadlines and processing times.

We are also interested in using a genetic algorithm to improve the quality of our solutions. A genetic algorithm is a heuristic search that has been used to find good solutions to a number of different optimization problems, but genetic algorithms searching for good schedules must overcome the difficulty of manipulating the sequences of jobs. We investigate the use of a genetic algorithm to search a new type of space, the problem space. This type of approach was introduced in Storer, Wu, and Vaccari (1992), who consider alternative search spaces for the

general job shop scheduling problem. In this work, we extend the idea to the problem of one-machine class scheduling.

Our search attempts to adjust the deadlines of the given problem so that our heuristic will find even better solutions. The space of adjusted deadlines that we search forms a *problem space* (we will return to this point in Section 3.2.5). The second contribution of this work is our use of this method to improve the scheduling of a single-machine problem.

If we use the principles of Davis (1991), then we can classify our genetic algorithm as a type of hybrid genetic algorithm. However, the only unusual characteristic of our algorithm is the decoding (the bit string does not describe a point in the solution space; instead it must be mapped to a solution via the heuristic). Moreover, the range of hybrid genetic algorithms is so large (for instance, Goldberg, 1989, describes hybrids differently) that our use of the term *problem space genetic algorithm* is a more precise description of the search. Finally, this problem space exists independently of the genetic algorithm, and the use of this new search space is not limited to our search. Other searches (including steepest descent, simulated annealing, and tabu search) could be used to explore the space. Therefore, we will continue to refer to our search space as a problem space and to our search as a problem space genetic algorithm.

The next subsection summarizes some of the relevant literature on class scheduling problems and the dual criteria objective under consideration. In Section 3.2.3, we discuss our notation, an example instance of the problem, and a number of basic results. We discuss in Section 3.2.4 the heuristic developed for the problem. Our genetic algorithm will employ this heuristic. In Section 3.2.5, we present a problem space, introduce genetic algorithms, and discuss the details of the genetic algorithm we developed to search the problem space. Section 3.2.6 describes the generation of the sample problems, the computational experiments, and the results. Finally, in Section 3.2.7, we present our conclusions.

3.2.2 Literature Review

In this section we will briefly mention some of the relevant research on class scheduling and on the dual criteria problem of minimizing total flowtime subject to job deadlines. This owrk and the literature on genetic algorithms are discussed in more detail in Chapter 2.

One of the first papers on problems with class scheduling characteristics is Sahney (1972), who considers the problem of scheduling one worker to operate two machines in order to minimize the flowtime of jobs that need processing on one of the two machines. Sahney derives a number of optimal properties and uses these to derive an branch-and-bound algorithm for the problem. Gupta (1984) defines the class scheduling problem, and Potts (1991), Coffman, Nozari, and Yannakakis (1989), and Ho (1992) also study two-class scheduling problems.

Bruno and Downey (1978) prove that, for more general class scheduling problems, the question of finding a schedule with no tardy jobs is NP-complete. Monma and Potts (1989) prove that many class scheduling problems are NP-complete, including minimizing makespan, maximum lateness, the number of tardy jobs, total flowtime, and weighted flowtime.

Dobson, Karmarkar, and Rummel (1987, 1989), Gupta (1988), Ahn and Hyun (1990), and Mason and Anderson (1991) all study the class scheduling problem under different objective functions. We will modify the procedure of Ahn and Hyun in order to use it as a comparative heuristic. The only other dual criteria problem in this area is studied by Woodruff and Spearman (1992); they consider a class scheduling problem with profit maximization and deadlines.

In the dual criteria literature, the problem of minimizing total flowtime subject to job deadlines (a deadline is a constraint on the completion time) is among the oldest questions, being first studied by Smith (1956). The problem of minimizing the weighted flowtime subject to job deadlines is a strongly NP-complete problem (Lenstra, Rinnooy Kan, and Brucker, 1977), and a number of researchers have examined branch-and-bound techniques.

3.2.3 Notation and an Optimal Property

In this section we introduce our problem and notation, give an example instance of the class scheduling problem under consideration, and present some basic results.

Our class scheduling problem is the minimization of the total flowtime of a set of jobs where the jobs have deadlines on their completion. The problem can be formulated with the following notation:

J_j	Job $j, j = 1, \ldots, n$
p_j	processing time of J_j
D_j	deadline of J_j
G_i	job class $i, i = 1, \ldots, m$
n_i	number of jobs in G_i
s_{0i}	time of initial setup if first job is in G_i
s_{ki}	time of setup between jobs in G_k and G_i
C_j	completion time of J_j
$\sum C_j$	total flowtime.

The problem is to find a sequence that minimizes the total flowtime ($\sum C_j$) subject to the deadline constraints ($C_j \leq D_j$ for all J_j). We name this problem the *Constrained Flowtime with Setups* problem (*CFTS*). Since job preemption or inserted idle time leads to a non-optimal solution, we will assume that schedules being considered have neither. Any schedule that is a solution for CFTS will have a number of *batches* or *runs* that are sets of jobs from one class processed consecutively. Before each batch will be a class setup. The problem involves determining the composition and order of batches from different classes.

CFTS is an extension of a one-machine problem studied by Smith (1956). In his problem, which we name the *Constrained Flowtime* problem (*CFT*), there exist no sequence-dependent setup times.

An instance that we will use to illustrate our work is described in Example 3.1.

Example 3.1. The data in Table 3.1 form an instance of a class scheduling problem with five jobs in two job classes. The first three jobs form one class, with the remaining two jobs in the

second class. Recall that no setup is required between jobs in the same class. However, a class setup is necessary between jobs of different classes.

Table 3.1. Job and Class Data for Example 3.1.

j	1	2	3	4	5
p_j	1	2	2	3	2
D_j	3	16	14	10	18
$G_1 = \{1, 2, 3\}$	$n_1 = 3$	$s_{01} = s_{21} = 2$			
$G_2 = \{4, 5\}$	$n_2 = 2$	$s_{02} = s_{12} = 1$			

CFTS is an NP-complete problem since finding a feasible schedule is NP-complete (Bruno and Downey, 1978). Hence, it is unlikely that any polynomial algorithm to solve the problem exists. We will study the use of heuristics to find good solutions.

We now describe Smith's rule for CFT and an optimal property for CFTS. Our new heuristic extends Smith's rule by taking advantage of the optimal property, which we call Smith's property.

In this and later sections, we will refer to each job that can complete at a given time without violating the job's deadline as being *eligible*. In this problem, we are concerned with deadlines that are constraints on the completion times, and we will create schedules backwards, starting with the last position in the sequence. Thus, we say that a job J_j is eligible at a time t if $t \leq D_j$. The job can feasibly complete at this time without violating the deadline constraint.

Smith's property for CFT states that if a job is assigned the last position of an optimal schedule, then it must be the longest eligible job. Smith's rule is derived from this property.

Algorithm 3.1 (Smith's rule for CFT). Let $t = p_1 + \ldots + p_n$. Among the jobs that are eligible at time t, that is, $t \leq D_j$, choose the job J_j with the longest processing time p_j. Schedule J_j to complete at t, and solve the remaining problem in a similar manner.

The following property extends Smith's rule to each class in CFTS. This property will then be extended to consider all classes in order to generate approximate solutions to CFTS.

Lemma 3.1 (Smith's property for CFTS). For each class in an optimal schedule for CFTS, the only job that could be scheduled to complete at a time t is the longest eligible one.

Proof. It suffices to show that if two jobs J_i and J_j in the same class are both eligible at time t and J_i is longer than J_j ($p_i > p_j$), then scheduling J_j to complete at time t leads to a non-optimal solution. Suppose we do. Then $C_j = t$, and J_i precedes J_j in the schedule formed. Create a new schedule by interchanging the two jobs. Since J_j is moved to the left, it is still feasible, and the new completion time is less than C_i, the old completion time of J_i ($p_i > p_j$). The completion times of any jobs between J_j and J_i are decreased. Meanwhile, J_i completes when J_j did, but this is feasible since $t \leq D_i$. We have therefore created a feasible schedule with less total flowtime, and the original schedule cannot be optimal. QED.

3.2.4 The Heuristic

Quick methods of finding good solutions are sometimes effective ways to attack difficult problems. In this section we describe a multiple-pass heuristic that extends the idea of Smith's rule. We illustrate how this heuristic works using Example 3.1.

Our heuristic finds solutions for CFTS by scheduling jobs in the spirit of Smith's rule, working backwards from the end of the schedule. Since the makespan (the maximum completion time) of the optimal solution is not known, the heuristic starts with a trial makespan. After scheduling all of the jobs, we compute the actual makespan (by removing any idle time) and use this makespan as the starting point for another iteration. We continue this process until some limiting makespan is reached. At this point, another pass of the heuristic yields a schedule with the same makespan or a schedule that is infeasible (because some job or setup starts before time zero).

This heuristic constructs schedules that satisfy Smith's property for CFTS (Lemma 3.1). While that lemma applies only to jobs in one class, our algorithm extends the idea of longest

eligible job by considering all of the job classes. We schedule the longest job with the minimum wasted time. Wasted time is time spent in a setup and idle time.

This (single-pass) Minimum Waste algorithm schedules an eligible job from the same class as the previously scheduled job if one exists. Else, it selects a job from the class with the smallest setup or selects the job with the latest deadline. It does this by measuring each job's *gap*: the wasted time incurred by selecting that job. Note that if no class setups exist, this algorithm is the same as Smith's rule for CFT (Algorithm 3.1).

Algorithm 3.2 (Single-pass) Minimum Waste.

Step 0: Given a completion time t, select for the last job the longest job eligible at this time J_j, i.e. $t \leq D_j$. Schedule this job to end at t, and reduce t by p_j.

Step 1 (a modification to Smith's rule): Suppose that at time t, a job from class G_i starts. Then, for each unscheduled job J_j, define q_j as the gap between the last possible completion time of J_j and t. If J_j is in class G_k, $q_j = \max \{t - D_j, s_{ki}\}$ (see Remarks below for an explanation of this definition). Let $q = \min \{q_j \text{ over unscheduled } J_j\}$. Select the longest job J_j with $q_j = q$ and schedule this job to end at $t - q$. Any necessary setup s_{ki} can begin at $t - q$. Reduce t by q and p_j.

Step 2: If there remain unscheduled jobs, return to Step 1.

Step 3: There are no more unscheduled jobs. If the first job in the schedule is in class G_i, a setup of length s_{0i} must end at t. Reduce t by this amount.

Step 4: If $t < 0$, the schedule created is infeasible. Else, compute the actual makespan of the jobs and setups scheduled.

Remarks. In order to motivate the definition of q_j, the gap, let us note that the setup after J_j is s_{ki}, so the job may not complete after $t - s_{ki}$. However, if the deadline $D_j < t - s_{ki}$, the gap will be $q_j = t - D_j$, which is greater than s_{ki}. This gap thus includes the setup and a period of inserted idle time of length $t - D_j - s_{ki}$. Note that if J_j is also in class G_i, $q_j = 0$ if and only if $t \leq D_j$. The algorithm is a type of greedy heuristic, in that it attempts to minimize the setup time or idle time in selecting jobs to be scheduled.

Multiple-Pass Minimum Waste Heuristic. To find a good solution for CFTS, we can use the following procedure that makes use of the single-pass Minimum Waste algorithm.

Step A: Let $t' = \max \{D_j : j = 1, \ldots, n\}$.

Step B: Let $t = t'$.

Step C: Perform one pass of the Minimum Waste algorithm (Algorithm 3.2) with completion time t. This creates a trial schedule.

Step D: Let t' be the sum of processing times and setup times of this schedule. If $t' < t$ and the trial schedule is feasible, go to Step B. (The smaller makespan may yield another schedule.)

Step E: If the trial schedule was infeasible or $t' = t$, take the last feasible schedule created and remove the inserted idle time, starting all jobs as soon as possible. This schedule is the result of the heuristic. If an infeasible schedule was created on the first pass, then take the sequence of jobs from the schedule and process the jobs in this order, starting at time zero. This will yield a schedule with some violated deadline constraints.

Because the problem of finding a feasible schedule is NP-complete, a single pass of the Minimum Waste algorithm is not guaranteed to find one. Still, as we shall see, it is usually able to find a feasible schedule if one exists. If a feasible schedule exists, it must finish by the maximum deadline, which is the first trial makespan. Initially, the heuristic is concerned with reducing the makespan. Eventually, as the makespan reaches a lower limit, the algorithm concentrates on the flowtime objective through its use of Smith's property to select a job.

Example 3.2. Let us apply the Multiple-Pass Minimum Waste heuristic to Example 3.1. In the first iteration, $t = 18$, the maximum deadline. The first pass of the Minimum Waste algorithm performs the following calculations (see Table 3.2 for complete algorithm): at time 18, no jobs have been scheduled, and the only eligible job is J_5. After choosing J_5, t is reduced by $p_5 = 2$ to 16. For J_1, J_3, and J_4, the waste is the gap until the deadline. Thus, $q_1 = 16 - D_1 = 13$, and similarly for the other two jobs. For J_2, however, $D_2 = 16$, and the deadline gap is zero, but because J_2 is in a different class than J_5, $q_2 = s_{12} = 1$. Thus, J_2 has the smallest waste and is scheduled to end at time 15. After five steps, all of the jobs are scheduled (see Figure 3.1). There

are two units of inserted idle time, however, so the actual makespan of the schedule can be reduced to 16.

Table 3.2. Calculations of the first pass of the Minimum Waste algorithm. Initial makespan = 18.

Time:	Waste: J_1	J_2	J_3	J_4	J_5	Schedule: J_j	C_j
18	15	2	4	8	0	J_5	18
16	13	1	2	6		J_2	15
13	10		0	3		J_3	13
11	8			2		J_4	9
6	3					J_1	3
Scheduled flowtime: 58. Reduced makespan: 16.							

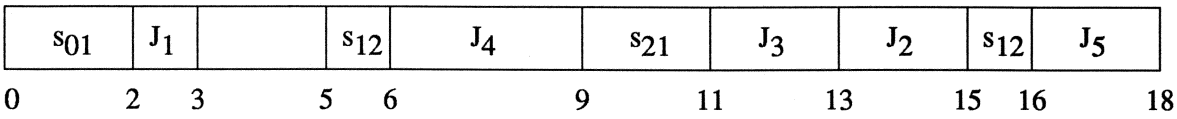

Figure 3.1. Schedule created on first pass of heuristic.

When the heuristic repeats the algorithm with the new makespan of 16 (see Table 3.3 for calculations), jobs J_2 and J_5 are eligible at time 16. These jobs also have the same processing time, but suppose J_2 is chosen. Then, at time 14, J_3 is eligible and has no gap, as it is in the same class as J_2. However, J_5 has a gap $q_5 = s_{12} = 2$. The algorithm continues in this manner, creating the schedule shown in Figure 3.2.

The Multiple-Pass Minimum Waste Heuristic again repeats the algorithm, which now begins at the reduced makespan of 15, and a similar schedule can be found if J_2 is chosen at time 15. This schedule has no idle time (see Figure 3.3), the reduced makespan is also 15, and so the heuristic stops.

Table 3.3. The second pass of the Minimum Waste algorithm. Initial makespan = 16.

Time:	Waste: J_1	J_2	J_3	J_4	J_5	Schedule: J_j	C_j
16	13	0	2	6	0	J_2	16
14	11		0	4	2	J_3	14
12	9			2	2	J_4	10
7	6				0	J_5	7
5	2					J_1	3
Scheduled flowtime: 50. Reduced makespan: 15.							

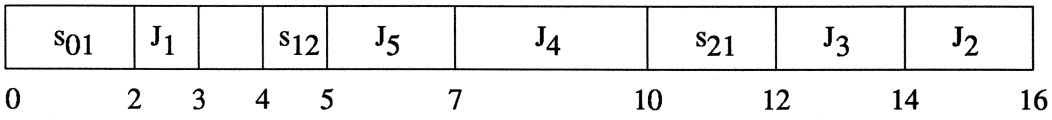

Figure 3.2. Second schedule.

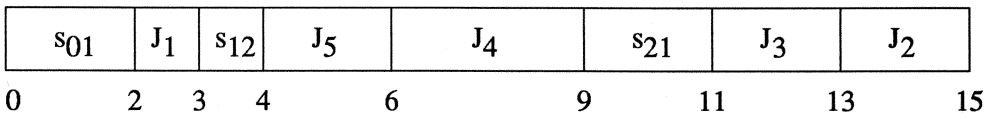

Figure 3.3. Third and final schedule.

3.2.5 The Genetic Algorithm

In this section we present an alternative search space, the problem space, provide a brief introduction to genetic algorithms and some references to selected works, and discuss the details of the problem space genetic algorithm we used to find good solutions for CFTS.

Problem space. The original work on problem and heuristic spaces is by Storer, Wu, and Vaccari (1990, 1992), who define some alternative search spaces for the job shop scheduling problem: the *problem space* and the *heuristic space*. They note that a solution to a problem is the result of applying a heuristic to the problem. Given a problem p, a heuristic h is a function that creates a sequence corresponding to a solution s, i.e. $h(p) = s$. Thus, if one adjusts the heuristic, one creates a different solution. The set of adjusted heuristics is the heuristic space. Likewise, if

one adjusts the problem data that are used by the heuristic, one generates a different solution. This set of adjusted problem data is the problem space. The idea is applied to the job shop scheduling problem, and different heuristic searches over the spaces are performed, including hill climbing, genetic algorithms, simulated annealing, and tabu search.

Our research extends this idea by defining a problem space for the one-machine class scheduling problem. If we adjust the deadlines that are inputs (along with the other problem data) to a pass of the Minimum Waste algorithm, we will create a possibly different schedule. We will use as a problem space for CFTS these *adjusted deadlines*, and we will use one pass of the Minimum Waste algorithm to create a sequence of jobs using the adjusted deadlines. The feasibility (against the actual deadlines) and total flowtime of the sequence can be evaluated by scheduling the jobs to start at time zero with no inserted idle time. The idea is to force jobs to be done earlier or later by decreasing or increasing the deadlines. We will prove that every solution for CFTS (including the optimal one) is in the range of h.

Theorem 3.1. For each solution to an instance of CFTS, there exists a vector of adjusted deadlines that can be mapped to that solution using one pass of the Minimum Waste algorithm.

Proof. Suppose that σ is a solution (a feasible schedule with no preemption or inserted idle time) for an instance of CFTS. For each job, consider adjusting the deadline so that it equals the job completion time. Then, if we use one pass of the Minimum Waste algorithm with the adjusted deadlines, the job selected for the last position will be the job with the maximum adjusted deadline. This job is the one with the maximum completion time C_i and thus was the last job in σ. It will be scheduled to complete at its adjusted deadline, which is C_i.

Now we are at the start time t of a job J_i and the job with the smallest gap is the unscheduled job J_j that immediately precedes J_i in σ, since the adjusted deadline is C_j, and $C_j \leq t$. Any setup necessary between J_j and J_i is already included in the difference between C_j and t. Thus the gap cannot be larger for this job, and the gap for any other job J_k is larger since $C_k < C_j$. This job will be scheduled to complete at its adjusted deadline, which is C_j. If we continue in this

manner, all of the jobs will be sequenced in the same order as they were in σ, and we create the same schedule. QED.

The more we adjust the deadlines, the more change we create in the schedule. For instance, consider the following examples of applying one pass of the Minimum Waste algorithm to vectors of adjusted deadlines where we have changed only the second and fifth deadlines: (Note that the fourth schedule created is infeasible since J_2 completes at time 18, which is greater than the actual deadline: $D_2 = 16$. The adjusted deadline of 19 was used only to sequence the jobs.)

Heuristic(problem) = solution:

Minimum Waste (3, 6, 14, 10, 20) = [1 2 4 3 5], flowtime 46.
Minimum Waste (3, 16, 14, 10, 18) = [1 4 3 2 5], flowtime 50 (original deadlines).
Minimum Waste (3, 17, 14, 10, 16) = [1 5 4 3 2], flowtime 46.
Minimum Waste (3, 19, 14, 10, 17) = [1 4 3 5 2], infeasible ($C_2 = 18$).

In Figure 3.4 we show a graph that illustrates how adjusting just two of the five deadlines of Example 3.1 can create a number of different schedules. The first, third, and fourth deadlines were not adjusted. Each point in the plane (only non-negative deadlines were considered) corresponds to a pair of values for the second and fifth adjusted deadlines. The points in each region of the plane are mapped by a pass of the Minimum Waste algorithm to the job sequence denoted by the five-digit sequence shown in that region. The dot marks the point that corresponds to the unadjusted deadlines ($D_2 = 16, D_5 = 18$). The best sequences achievable by adjusting these deadlines are 12435 and 15432 (total flowtime = 46), and the only other feasible sequences are 14235 and 14325 (total flowtime = 50). The optimal solution (which cannot be found by adjusting only the second and fifth deadlines) is 13452, with total flowtime = 43.

Since the actual problem space consists of all of the problem data and there are numerous heuristics that can be used, we can investigate other spaces and heuristics that might be useful. Our first search was to adjust the job processing times and to use the Shortest Processing Time (SPT) rule. However, it is difficult to find feasible solutions since SPT ignores the deadline constraints entirely. We also tried the using the Earliest Due Date (EDD) rule while adjusting the

deadlines, but EDD does not give enough attention to the flowtime objective. Sequencing by either SPT or EDD is a fairly naive heuristic, since neither makes use of the other available information. The Minimum Waste algorithm, however, considers due dates, processing times, and setups, and using it improves our searches. In addition, while it would be possible to use the Minimum Waste algorithm while adjusting the processing or setup times, the effect of these variables on the sequencing of jobs is more indirect than that of the deadlines.

The use of a heuristic space seems to be hard for this problem. Feasibility is a large concern, and there are very few heuristics we can use to find feasible schedules. Also, the Minimum Waste algorithm has no parameters to adjust.

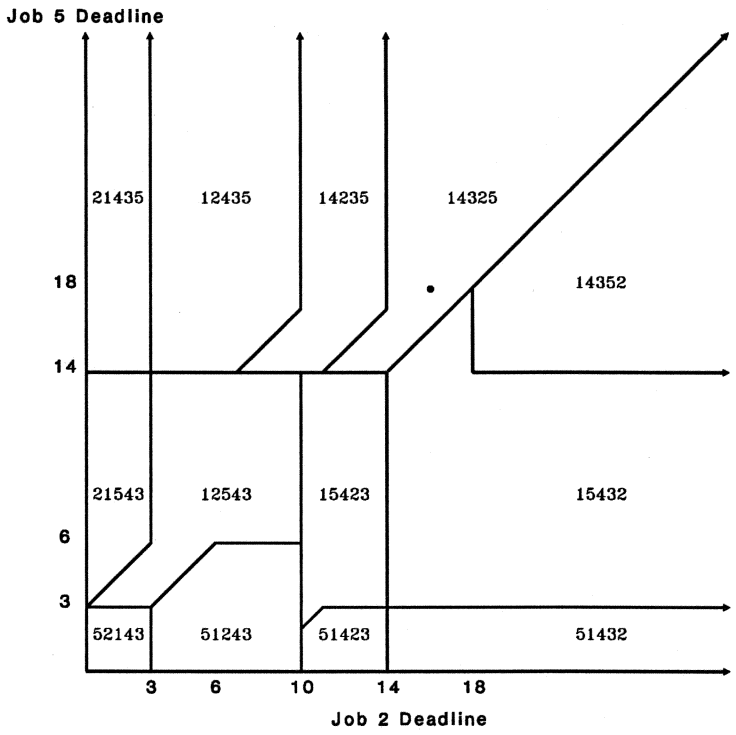

Figure 3.4. Graph of adjusted deadlines and schedules created.

A genetic algorithm for CFTS. In this section we discuss the details of the genetic algorithm we developed to find better solutions for CFTS. As mentioned before, genetic algorithms are heuristic searches that use a population of points in the effort to find the optimal solution. The stronger members of the population survive, mate and produce offspring that may undergo a mutation. These offspring form a new generation. Genetic algorithms have been used on sequencing problems before, although they cannot use natural crossover techniques in searching the solution space. The advantage of the problem space is that the genetic algorithm can use standard techniques to create offspring.

Our genetic algorithm searches the space of adjusted deadlines. We use a binary coding for the adjusted deadlines and a single pass of the Minimum Waste algorithm as the heuristic. For this genetic algorithm, we use many of the ideas presented in Davis (1991), to which we refer readers who wish to learn more about the issues discussed here.

In the problem space, each point is a vector of integers that are deadlines used as input for one pass of the Minimum Waste algorithm. We will use a binary representation of the points in problem space. In the population of the genetic algorithm, each individual is a string of bits. Each successive six-bit substring represents a deadline for a specific job. The integer decoded from this binary number ranges from zero to 63 and linearly maps to a real number in the range from zero to the maximum deadline in the given problem data. This discretization reduces the problem space but still allows the deadlines to vary significantly with respect to each other.

The adjusted deadlines are used as input to a single pass of the Minimum Waste algorithm, which outputs a sequence of jobs. The algorithm uses the largest of the adjusted deadlines as the initial makespan and schedules the jobs accordingly, using the actual job processing and class setup times where necessary but using the adjusted deadlines to determine when a job is eligible. If necessary, the algorithm can start jobs before time zero.

Using the actual problem data and the sequence of jobs output from one application of the Minimum Waste Algorithm, we can create a schedule of jobs that starts at time zero and has no

Table 3.4. Bit representation of the dummy.

$\lfloor 63\ D_j/18 \rfloor$	10	56	49	35	63
Bits	001010	111000	110001	100011	111111

Table 3.5. A point in the problem space.

Bits	001010	111000	110001	100111	011111
Integer	10	56	49	39	31
D_j	2.85	16	14	11.14	8.85

Table 3.6. Application of the Minimum Waste algorithm.

Time:	Waste: J_1	J_2	J_3	J_4	J_5	Schedule: J_j	C_j
16	13.15	0	2	4.86	7.15	J_2	16
14	11.15		0	2.86	5.15	J_3	14
12	9.15			2	3.15	J_4	10
7	4.15				0	J_5	7
5	2.15					J_1	2.85

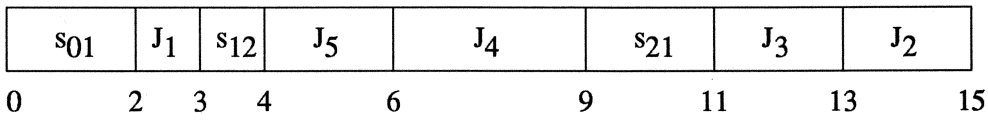

Figure 3.5. Schedule corresponding to new bit string.

After some experimentation we decided to use a steady-state genetic algorithm that created offspring by repeatedly (and randomly) selecting from a set of four genetic operators: one-point crossover, uniform crossover, and two types of mutation. *Small mutation* used a low probability (two percent per bit) of flipping a bit in the string; *large mutation* used a higher probability (fifty percent). The search used tournament selection for selecting the parents necessary for the crossover or mutation, and duplicate bit strings were not allowed in the population. Tuning was performed in order to determine some good settings for various algorithm parameters. (See Table 3.7. The problem sets used for tuning are described in Section 3.2.6.)

Table 3.7. Flowtime performance while tuning 2000-individual genetic algorithm on 30-job problems (Newprob4).

Population Size	Operator Probabilities (in percent)	Increase in *r*	Frequency of Increase	Mutation Rate	Average Ratio
Tuning Increase in *r* and Frequency of Increase, Population Size = 10					
10	25, 25, 25, 25	+2	10	2%	0.8575
10	25, 25, 25, 25	+10	10	2%	0.8666
10	25, 25, 25, 25	+50	10	2%	0.8562
10	25, 25, 25, 25	+2	50	2%	0.8646
10	25, 25, 25, 25	+10	50	2%	0.8700
10	25, 25, 25, 25	+50	50	2%	0.8624
10	25, 25, 25, 25	+2	100	2%	0.8676
10	25, 25, 25, 25	+10	100	2%	0.8755
10	25, 25, 25, 25	+50	100	2%	0.8625
Tuning Increase in *r* and Frequency of Increase, Population Size 50					
50	25, 25, 25, 25	+2	10	2%	0.8597
50	25, 25, 25, 25	+10	10	2%	0.8660
50	25, 25, 25, 25	+50	10	2%	0.8623
50	25, 25, 25, 25	+2	50	2%	0.8669
50	25, 25, 25, 25	+10	50	2%	0.8687
50	25, 25, 25, 25	+50	50	2%	0.8654
50	25, 25, 25, 25	+2	100	2%	0.8619
50	25, 25, 25, 25	+10	100	2%	0.8626
50	25, 25, 25, 25	+50	100	2%	0.8669
Tuning Increase in *r* and Frequency of Increase, Population Size 100					
100	25, 25, 25, 25	+2	10	2%	0.8800
100	25, 25, 25, 25	+50	10	2%	0.8794
100	25, 25, 25, 25	+2	50	2%	0.8738
100	25, 25, 25, 25	+50	50	2%	0.8792
100	25, 25, 25, 25	+2	100	2%	0.8751
100	25, 25, 25, 25	+50	100	2%	0.8716
Tuning Mutation Rate					
10	25, 25, 25, 25	+50	10	2%	0.8562
10	25, 25, 25, 25	+50	10	10%	0.8990
10	25, 25, 25, 25	+50	10	50%	0.9962
10	25, 25, 25, 25	+50	50	2%	0.8624
10	25, 25, 25, 25	+50	50	10%	0.8953
10	25, 25, 25, 25	+50	50	50%	0.9629
50	25, 25, 25, 25	+50	10	2%	0.8623
50	25, 25, 25, 25	+50	10	10%	0.8994
50	25, 25, 25, 25	+50	10	50%	0.9236

Table 3.7. (Continued)

Population Size	Operator Fitnesses (in percent)	Increase in r	Frequency of Increase	Mutation Rate	Average Ratio
Tuning Operator Fitness					
10	75, 0, 0, 25	+50	10	2%	0.8664
10	50, 15, 10, 25	+50	10	2%	0.8764
10	25, 25, 25, 25	+50	10	2%	0.8562
10	15, 50, 10, 25	+50	10	2%	0.8758
10	0, 75, 0, 25	+50	10	2%	0.8592
10	75, 0, 0, 25	+50	50	2%	0.8730
10	50, 15, 10, 25	+50	50	2%	0.8628
10	25, 25, 25, 25	+50	50	2%	0.8624
10	15, 50, 10, 25	+50	50	2%	0.8649
10	0, 75, 0, 25	+50	50	2%	0.8688
50	75, 0, 0, 25	+50	10	2%	0.8574
50	50, 15, 10, 25	+50	10	2%	0.8732
50	25, 25, 25, 25	+50	10	2%	0.8623
50	15, 50, 10, 25	+50	10	2%	0.8689
50	0, 75, 0, 25	+50	10	2%	0.8720
Tuning 3000-individual genetic algorithm on 50-job problems (Prob50z).					
10	25, 25, 25, 25	+50	50	2%	0.8880
50	25, 25, 25, 25	+50	50	2%	0.8931
50	25, 25, 0, 50	+50	50	2%	0.8739

Notes: Performance is average ratio to solution found by the Multiple-Pass Minimum Waste Heuristic. Operators are one-point crossover, uniform crossover, large mutation, and small mutation. Mutation rate is probability per bit.

These parameters included the population size, the mutation rate, the operator selection probabilities, and the rate of change of the penalty coefficient r. The rate of increase in the penalty coefficient affected the solution quality slightly. The mutation rate, population size, and relative probabilities of operator selection affected solution quality more significantly, with a smaller mutation rate (two percent, as mentioned earlier), smaller population sizes, and a higher probability for selecting the small mutation yielding better solutions. These factors imply that the search can easily find good neighborhoods (especially since a point corresponding to the original problem data is included in the initial population) but needs to spend time hunting for a better

solution. Thus, a search that incorporates some kind of local search at the end of the genetic algorithm may be useful.

3.2.6 Empirical Testing

In this section we describe a set of experiments performed in order to test the genetic algorithm and the heuristics. We discuss the generation of sample problems and the computational results.

Problem generation. In order to test the heuristics and the genetic algorithm described above, it is necessary to create a set of test problems. We describe in this section how we can create problems that have at least one feasible solution and problems where finding a feasible solution is more difficult.

The problems in the first problem set have 30 jobs in four classes, with random processing times in the range [1, 20] and sequence-dependent setup times in the range [0, 5]. For this set, we want to determine random deadlines in order to insure that some feasible schedule did exist.

We use the following procedure: after computing the random class setup times, each job is given a random processing time, and an initial completion time is computed by scheduling it after all previously constructed jobs. This first-generated, first-served schedule yields a makespan that becomes an upper bound for the deadlines, and each job is given a deadline determined by sampling a random variable uniformly distributed between the job completion time (in this schedule) and the makespan, i.e. the interval [C_j, C_{max}]. Thus, the initial sequence is a feasible solution.

In order to determine the performance of the genetic algorithm on minimizing the flowtime when feasible solutions are harder to locate, a number of additional problem sets are created. In addition to the problem set described earlier, which includes problems that were known to have a feasible solution, we generate 30-job and 50-job problems with tighter deadlines. The 30-job problems have four job classes and the 50-job problems ten job classes. Tighter deadlines are achieved by extending the range of values that a random deadline could take. Let us define a value k that can range from zero to one. The deadline for J_j is taken from the interval

[$k\ C_j$, C_{max}], where the C_j and C_{max} are from the original generated schedule. If $k = 1$, this plan is the same as the original one, and the generated problem is guaranteed to have a feasible schedule. If $k = 0$, all of the deadlines vary equally, and there may exist no feasible schedule. As k decreases from one to zero, the problems we generate have a higher probability of having fewer feasible schedules. We generated problems with $k = 0$, 0.2, and 1. Since the Multiple-Pass Minimum Waste Heuristic cannot find feasible solutions for some of these problems, we will see if the genetic algorithm can find solutions that are feasible.

Results. In this section we discuss the results of our experiments with the solution procedures on the generated problem sets. We summarize the findings and present tables of the collected data. Since the optimal solutions are not known (a branch-and-bound algorithm to find optima requires excessive computational time) and no good lower bound can be determined, we measure the performance of the solution procedures relative to each other.

Each procedure was run once on each of the problems in the problem sets. The procedures include the Multiple-Pass Minimum Waste Heuristic and the problem space genetic algorithm.

For comparison purposes, we also implemented a version of the heuristic that Ahn and Hyun (1990) use to reduce the total flowtime in class scheduling problems. They proposed an iterative heuristic that starts with an initial feasible sequence where the jobs in each class are in SPT order (since they are not concerned with deadlines) and applies both a forward and backward procedure to it, repeating the steps until no strict improvement is found. Each of the forward and backward procedures interchanges different subschedules where the second subschedule consists of jobs from one class and the first subschedule has no jobs from this class. If the interchange reduces the total flowtime, the subschedules are switched; this maintains the class SPT property.

Our version of this algorithm, called the Modified Ahn & Hyun heuristic, uses one pass of the Minimum Waste algorithm to form the initial schedule. In addition, a potential swap of two subschedules is performed only if the swap reduces the total flowtime and maintains deadline feasibility.

The results (see Table 3.8) show that the genetic algorithm can find solutions that are much better than those found by the Multiple-Pass Minimum Waste Heuristic and are slightly better than those that Modified Ahn & Hyun heuristic produces. The genetic algorithm needs more time to find good solutions on the larger problems, although additional tuning may help improve the performance of the search.

The searches for the 30-job problems were for 2000 iterations using the following parameter settings: population size of 10, all operator fitnesses equal, increase of 50 in r every 10 individuals. The searches for the 50-job problems were for 3000 iterations using the same population size, no large mutations, and an increase of 50 in r every 50 individuals.

Table 3.8. Total flowtime performance of heuristics on problems where a feasible was found.

Problem Set	Problems	Jobs	k	Performance[a] of Heuristics	
				Genetic Algorithm	Modified Ahn & Hyun
Newprob4	10	30	1	0.8562	0.9130
Prob2a	4	30	0.2	0.9099	0.9346
Prob50z	10	50	1	0.8739	0.8755
Prob2c	8	50	0.2	0.8796	0.8914

Note: *a*: Performance is average ratio to solution found by the Multiple-Pass Minimum Waste Heuristic.

We observed that if the Multiple-Pass Minimum Waste Heuristic is unable to generate a feasible solution, the Modified Ahn & Hyun heuristic and the problem space genetic algorithm cannot locate a feasible solution. Thus our results are reported only for those problems where feasible schedules were created.

The genetic algorithm is able to outperform the Multiple-Pass Minimum Waste heuristic on total flowtime at the cost of increased computational time, as a full 2000-individual run lasts 256 seconds on average. (All computations performed on a 386 personal computer.) This is due to the effort of decoding the long bit strings into deadlines and the complexity of using one pass of the Minimum Waste algorithm to evaluate the individual. This computational effort is substantial

compared to that of running the Multiple-Pass Minimum Waste and Modified Ahn & Hyun heuristics. The heuristics need less than one second to find a good solution to a problem. A faster computer, however, would be able to reduce the computation time necessary for the genetic algorithm. Still, these results show that genetic algorithms that search the problem space can find very good solutions to scheduling problems.

The improvement in total flowtime of the solutions that the genetic algorithm can find is a result of two things: the multiple sampling of the search space and the evolutionary process. This leads to the following question: Are the genetic characteristics of the search a significant factor? We answer this question by changing the genetic algorithm so that each individual is a completely new one. Instead of choosing parents and creating offspring, we create new individuals by again mutating the dummy individual. This is a random sampling approach. (See Table 3.9.) These results imply that even though the same number of individuals are evaluated, the random sampling performs well but does not generate the same quality of solutions that the genetic algorithm does. Thus, we feel that the evolutionary process does contribute significantly to the improvement in total flowtime.

Table 3.9. Total flowtime performance of random sampling on problems where a feasible was found.

Problem Set	Problems	Jobs	*k*	Performance[a] of Random Sampling
Newprob4	10	30	1	0.9435
Prob2a	4	30	0.2	0.9545
Prob50z	10	50	1	0.9728
Prob2c	8	50	0.2	1.0165

Note: *a*: Performance is average ratio to solution found by the Multiple-Pass Minimum Waste Heuristic.

In conclusion, the genetic algorithm can find solutions with low total flowtimes. This good performance is due to the multiple sampling of the problem space (since more than one neighborhood can be searched at once); the use of the Minimum Waste algorithm to create

solutions from adjusted deadlines; and the ability of the genetic algorithm to combine the best characteristics of the points in the initial population.

3.2.7 Conclusions

This portion of the work has two contributions: it introduces an extended heuristic for the dual criteria class scheduling problem that we call CFTS, and it describes a problem space genetic algorithm used to find good solutions. The problem is to minimize the total flowtime subject to deadline constraints. In this section we present a multiple-pass heuristic for finding good solutions and discuss problem space and the genetic algorithm. Finally, we describe our experimental results, in which we compared the genetic algorithm to some heuristic approaches. From these results we make the following conclusions:

The Multiple-Pass Minimum Waste heuristic performs well at minimizing the total flowtime of CFTS. Though not an exact procedure, it is usually able to find feasible, high-quality solutions.

A genetic algorithm that searches a problem space of the Minimum Waste algorithm for CFTS can find solutions with lower total flowtime. This genetic algorithm includes a penalty function for infeasible points that increases the cost of tardiness as the search progresses. In addition, it produces slightly better solutions than another procedure modified for this problem.

3.3 Class Scheduling with Release and Due Dates

In this section, we study the one-machine class scheduling problem of minimizing the number of tardy jobs. Moreover, some of the jobs have non-zero release dates. We describe an extended heuristic developed for this problem and a genetic algorithm used to find good solutions. We also discuss an extension of this problem to the question of minimizing tardiness with minimum number of tardy jobs.

3.3.1 Introduction

The class scheduling problem studied in this section is to schedule a set of jobs, where some jobs have non-zero release dates, in order to minimize the number of tardy jobs. This problem is motivated by the semiconductor test area. Since post-assembly testing is the last stage in semiconductor manufacturing, meeting a job's due date is a very important objective for the manager of a test facility. The consideration of release dates is an attempt to model the look-behind situation that exists in the job shop, where the scheduling of a machine (the bottleneck, for instance) may be improved by including information about the jobs that are arriving soon.

This problem, like most class scheduling problems, is a difficult case. Since even finding a schedule with no tardy jobs is an NP-complete problem, exact algorithms to solve our problem in polynomial time do not exist. Thus, we are motivated to try different heuristics and searches.

Our approach was to modify an existing algorithm to include class setups and see how such an algorithm performs on this problem. Since the heuristic was not guaranteed to find good solutions, we also investigated a genetic algorithm. Thus, this research presents contributions in the extension of class scheduling problems to include a problem that has not been previously investigated and the use of both genetic algorithms and problem spaces to include the search for good solutions to class scheduling problems.

3.3.2 Literature Review

In this section we will mention some of the most relevant research on class scheduling and on the problem of minimizing the number of tardy jobs in the presence of release dates. A full discussion can be found in Chapter 2.

Bruno and Downey (1978) prove that, for general class scheduling problems, the problem of finding a schedule with no tardy jobs is NP-complete. Monma and Potts (1989) prove that class scheduling to minimize the number of tardy jobs is an NP-complete problem.

As discussed in Chapter 2, the one-machine problem of minimizing the number of tardy jobs when some have non-zero release dates ($1 / r_j / \sum U_j$) is a strongly NP-complete problem (Lawler, 1982). A restricted version of the problem has been considered by Kise, Ibaraki, and Mine (1978), who solve the problem optimally if the release and due dates match ($r_j < r_k$ implies $d_j \leq d_k$). They present an $O(n^2)$ algorithm (Kise's algorithm, described in Section 3.3.4) for this case.

3.3.3 Notation and Problem Formulation

We will use the basic notation introduced in Section 3.2.3. For CSRDD, each job J_j has a release date r_j and a due date d_j. For a given schedule, $U_j = 1$ if $C_j > d_j$ and 0 otherwise. The problem is to find a sequence that minimizes $\sum U_j$ subject to the constraint that $C_j \geq r_j + p_j$.

An NP-complete problem, CSRDD is unstudied in the literature on class scheduling. In order to simplify the problem, it is assumed that the release and due dates match; that is, there exists an ordering where the jobs are simultaneously in Earliest Release Date (ERD) order and in Earliest Due Date (EDD) order. Our primary heuristic for CSRDD extends Kise's algorithm for the problem without setups to form a heuristic for finding good solutions.

3.3.4 Heuristics

In this section we will describe a number of heuristics: Kise's algorithm for the problem without class setups, our extension of this algorithm, and other heuristics used for testing purposes.

Kise's algorithm. Kise's algorithm orders the jobs by their release and due dates (a non-ambiguous ordering since the dates must match). The algorithm is an extension of the Moore-Hodgson algorithm (Moore, 1968) for minimizing the number of late jobs. Each job is scheduled after the partial schedule of on-time jobs while maintaining release date availability. If the new job is tardy, the algorithm searches the on-time jobs for the job whose removal leaves the shortest schedule of on-time jobs. The removed job is made tardy and will be processed with the other

tardy jobs after the feasible jobs. In this manner, the algorithm finds the largest subset of the jobs that can be delivered on-time. These jobs are scheduled in order of their release and due dates. The search subalgorithm has effort that is linear in the number of jobs in the partial schedule. Since the subalgorithm may be performed up to n times, the total effort of Kise's algorithm is $O(n^2)$.

Kise's heuristic is not optimal for CSRDD, although it can be modified to include setup times. Take the following example:

Example 3.3.

j	r_j	p_j	d_j	i
1	0	5	6	1
2	0	4	13	2
3	5	5	14	2
4	6	2	15	1

$s_{01} = s_{02} = 1.\ s_{12} = 1.\ s_{21} = 4.$

The optimal sequence is $[J_1 J_4 J_2 J_3]$, with $C_1 = 6$, $C_4 = 8$, $C_2 = 13$, and $C_3 = 18$, which has one tardy job. Kise's algorithm adds J_3, which is tardy after J_1 and J_2, and the subalgorithm makes J_1 a tardy job. When J_4 is added to the schedule after J_2 and J_3, it also is tardy, for a total of two tardy jobs.

Kise extension. Our algorithm for CSRDD extends Kise's algorithm by considering two options when adding a new job to a partial schedule: we can place the job in a position after all of the on-time jobs or in a position after the last on-time job from the same class (if there is one). In either case, if an on-time job becomes tardy, we make tardy the job whose removal creates the shortest partial schedule of on-time jobs. We then choose between the two partial schedules created, selecting the new partial schedule with the smaller number of late jobs (or smaller makespan if they tie) as the incumbent before trying to schedule the next job.

Intuitively, it appears that the extended algorithm should outperform the Kise algorithm, since it includes an additional scheduling choice. Due to the complexity of the problem, however, this is not guaranteed. The following problem is one counter-example:

Example 3.4.

j	r_j	p_j	d_j	i
1	0	3	5	2
2	0	3	13	1
3	6	3	14	2
4	14	3	17	2

$s_{01} = s_{21} = 1$. $s_{02} = s_{12} = 2$.

The optimal sequence is [$J_1 J_2 J_3 J_4$], with $C_1 = 5$, $C_2 = 9$, $C_3 = 14$, and $C_4 = 17$, with none tardy. Kise's algorithm will construct this schedule. In the proposed algorithm, the addition of J_3 to [$J_1 J_2$] creates a partial schedule with no tardy jobs and a makespan of 14. The scheduling of J_3 after J_1 and before J_2 creates a makespan of 13 ($C_1 = 5$, $C_3 = 9$, $C_2 = 13$), so the sequence [$J_1 J_3 J_2$] replaces [$J_1 J_2 J_3$]. When we add J_4 after J_2, J_4 is tardy ($C_4 = 18$), and the on-time jobs complete at time 13; when J_4 is scheduled before J_2, J_2 is tardy ($C_4 = 17$, $C_2 = 21$). The algorithm thus yields [$J_1 J_3 J_2 J_4$], which is not an optimal schedule.

Tardiness rules. We also tested two heuristics based upon the R & M procedure of Rachamadugu and Morton (1982) and used for reducing weighted tardiness in Morton and Ramnath (1992). These are primarily class scheduling extensions.

RM: A dispatching rule where the priority of a job at time t is based upon the weight, the processing time, and the slack of the job:

$$\text{RM} = w_j / p_j * \exp(- S_j^+ / k * p_{avg}),$$

where RM is the job priority, w_j the job weight, p_j the processing time, S_j^+ the slack max $\{0, d_j - t - p_j\}$, k a predefined constant, and p_{avg} the average processing time of the jobs in the queue. In this formulation, jobs with higher weights, shorter processing times, and less slack will be scheduled first. According to Ramnath and Morton, the constant k is normally set to 2.

X-RM: The x-dispatch (look-behind) version of the RM rule includes jobs that will be arriving soon in an extended queue. That is, they arrive before the completion time of the shortest job already waiting. The RM priority is discounted by an amount that depends upon the arrival time:

$$\text{X-RM} = \text{RM} * (1 - (1.3 + p)*(r_j - t)^+/p_{min}),$$

where X-RM is the job priority, p the utilization factor, r_j the arrival time of the job, and p_{min} the smallest processing time among jobs currently available. Morton and Ramnath (1992) claim that this procedure reduces weighted tardiness by 40% over the standard RM rule.

These two priorities are used as dynamic dispatching rules. At a time t, the job with the highest priority is scheduled next. For our class scheduling problem, we redefine the components to include the setup times, but otherwise we use the same formulas. This works well for the objective of weighted tardiness, but for our objective (minimizing the number of tardy jobs), we would like to postpone the processing of the tardy jobs in order to concentrate on the on-time jobs. Thus, when we schedule a job that will be tardy, we look for the job whose removal will result in a shorter schedule with no tardy job and we remove that.

For our class scheduling problem, we include in the processing time the class setup necessary to process a job and modify the release date by the same amount. That is, if the job completing at time t is in G_i and J_j is in G_k, we add s_{ik} to p_j and subtract s_{ik} from r_j (for the X-RM calculation). We use $p = 1$ and $k = 2$ and $w_j = 1$ for all J_j.

3.3.5 Analysis of the Heuristic

In this section we discuss the computational effort necessary to perform the extended Kise heuristic and the worst case error of this heuristic. A pseudo-code presentation of the heuristic and its subalgorithm can be found in the Appendix.

It is obvious that the extended version of Kise's heuristic takes more effort than Kise's algorithm. However, the effort of the algorithm is still $O(n^2)$. When adding the next job to a partial schedule, there are two positions for the new job. The algorithm would take at most $O(n)$ effort to find the last job from the same class as the new job, to insert the new job, and to determine which (if any) jobs are now tardy. If there is a tardy job, a pass of Kise's subalgorithm must be performed to determine which job to remove. This is also $O(n)$. For the other position, there is $O(n)$ effort in adding the new job to the end of the partial schedule and performing a pass

of the subalgorithm. Thus, the total effort of adding the new job is O(n), and since n jobs must be scheduled, the total effort of the extended Kise heuristic is O(n^2).

Since CSRDD is strongly NP-complete, there is no optimal polynomial or pseudo-polynomial algorithm. Since our extended Kise heuristic is not guaranteed to find an optimal solution, we need to look into the worst-case error bound. In the following we describe two families of instances for CSRDD where the extended Kise heuristic cannot find good solutions. While the first of these examples prove that the heuristic can perform arbitrarily badly, the examples will also provide problem instances that we can use for testing the performance of the genetic algorithm. They are especially good for this since we know the optimal solution in advance.

Example 3.5. In this case, the extended Kise heuristic finds n - 1 out of n jobs tardy when the optimal has only two tardy jobs. There are n jobs where J_1 is in G_1 and $J_2, \ldots, J_n$ are in G_2, and the jobs have the following characteristics:

J_j	J_1	$J_2, \ldots, J_n$
r_j	0	1
p_j	1	1
d_j	1	n

The class setups are as follows:

G_i	G_1	G_2
s_{0i}	0	2
s_{1i}	-	n
s_{2i}	0	-.

The optimal sequence (for $n > 2$) begins with the n - 1 jobs in G_2. The last job ends at 1 + (n - 1) = n, so they are all on-time. Since J_1 is scheduled after this, it is tardy. The Kise and Kise-extension algorithms start by scheduling J_1 first. This job completes at time 1. When the first job from G_2 is scheduled to form [J_1 J_2], the job completes at 1 + n + 1 = n + 2 and is therefore tardy. Removing J_1 yields a partial schedule that ends at time 2 and is thus longer than the partial

schedule consisting of just J_1. Thus J_2 is made tardy. This continues for all of the jobs from G_2, and they are all forced to be tardy, for n - 1 tardy jobs.

Example 3.6. In this case the optimal solution has no tardy jobs and the extended Kise heuristic finds $n/3$ tardy jobs. We construct the problem instance in the following way: choose a non-negative integer k. Let $n = 3(k + 1)$. Let $m = 3$. For $i = 0, \ldots, k$, construct three jobs, J_{3i+1} in G_1, J_{3i+2} in G_2, and J_{3i+3} in G_3, with the following job characteristics, where $0 < \varepsilon < 1$ and $0 < \delta < 1$:

J_j	J_{3i+1}	J_{3i+2}	J_{3i+3}
r_j	3i	3i	3i
p_j	1-ε	1+ε	1
d_j	3i+2	3i+2+δ	3i+3.

Let the class setups be as follows, where $s > \delta$:

G_i	G_1	G_2	G_3
s_{0i}	0	0	0
s_{1i}	-	s	0
s_{2i}	0	-	s
s_{3i}	1	0	-.

The optimal sequence of jobs is [$J_2 J_1 J_3 J_5 J_4 J_6 \ldots J_{3k+2} J_{3k+1} J_{3k+3}$]. This schedule has no inserted idle time and no setup time (see Figure 3.6).

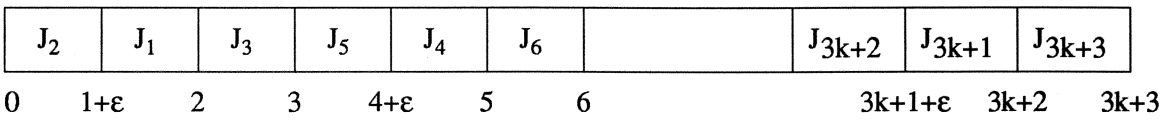

Figure 3.6. Optimal Schedule.

Taking the jobs in ERD order, we schedule J_1 first ($C_1 = 1 - \varepsilon$). J_2 is scheduled next, but $C_2 = C_1 + s_{12} + p_2 = 2 + s > d_2 = 2 + \delta$. Thus, J_2 is tardy, and since $p_2 > p_1$, J_1 remains while J_2 is now tardy. J_3 is added next, with $C_3 = C_1 + s_{13} + p_3 = 2 - \varepsilon$. J_4 is added next, starting at time $3 > C_3 + s_{31} = 3 - \varepsilon$ and completing at time $C_4 = 4 - \varepsilon$. (Starting J_4 after J_1 makes J_3 tardy.) By repeating the above argument, it can be shown that J_5 will be made tardy and J_6 will follow J_4.

This process continues for all of the jobs (see Figure 3.7). Thus, the heuristic creates a schedule where the $k + 1 = n/3$ jobs in G_2 are tardy, while the optimal schedule has no tardy jobs. Note that we never insert a job into the middle of a partial schedule; thus, Kise's algorithm and the extended Kise heuristic create the same schedule.

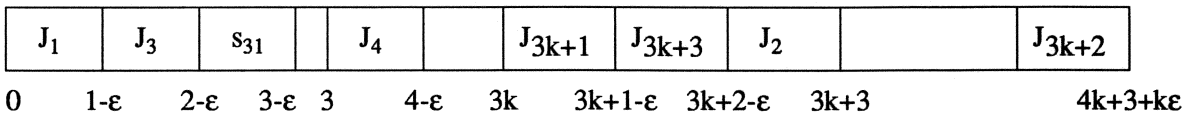

Figure 3.7. Heuristic Schedule.

The first of the two above cases will be useful in the proof of the following error bound theorem. For the purposes of Theorem 3.2, we assume that the triangle equality holds for the class setups: $s_{ab} + s_{bc} \geq s_{ac}$, for all $a = 0, \ldots, m, b = 1, \ldots, m, c = 1, \ldots, m$.

Theorem 3.2. For an instance of CSRDD, if there exists a schedule in which at least one job completes on-time, the extended Kise heuristic will create a schedule with at least one on-time job. Moreover, there exist problem instances where the heuristic will schedule exactly one on-time job, although there exist schedules with more than one on-time job.

Proof. First, we note that the number of scheduled on-time jobs never decreases as the extended Kise heuristic schedules new jobs. Now, consider a job J_j in class G_i that completes on-time in some feasible schedule. Thus, for J_j, $C_j \leq d_j$. Now, $C_j \geq r_j + p_j$ and $C_j \geq s_{0i} + p_j$, since by the triangle inequality stated above, if there exist any jobs before J_j, the sum of the setups before J_j is at least s_{0i}. Thus, $d_j \geq r_j + p_j$, and $d_j \geq s_{0i} + p_j$.

Suppose that the schedule created by the extended Kise heuristic has no on-time jobs. Then, when the extended Kise heuristic considered J_j, no other on-time jobs were scheduled. The heuristic started J_j as soon as possible (the maximum of r_j and s_{0i}), but J_j was tardy. Thus, $C_j = \max\{r_j, s_{0i}\} + p_j > d_j$, but this contradicts the result above. Thus, the heuristic creates a schedule with at least one on-time job.

Thus, we have shown that the extended Kise heuristic will schedule at least one on-time job. This bound is tight, as our discussion of Example 3.5 shows that there exist problems for which the heuristic will schedule exactly one on-time job although the optimal schedule has more than one on-time job.

3.3.6 The Genetic Algorithm

In this section we present the problem space and discuss the details of the genetic algorithm we used to find good solutions for CSRDD.

Problem space. In Chapter 2 we described the ideas of alternative search spaces. In this section we present the problem space that we searched in order to find good solutions for CSRDD.

We defined a problem space for CSRDD in the following manner: Given a problem p in problem space, a heuristic h is a function that creates a sequence corresponding to a solution s for CSRDD, i.e. $h(p) = s$. We defined a problem as a vector of job release dates, using a pass of Kise's algorithm to create a sequence of jobs by considering the jobs in order of their new release dates instead of the order imposed by the matching release and due dates. The actual release dates are used in determining the schedule, however. Note that all solutions for CSRDD (including the optimal ones) that schedule all tardy jobs last are in the range of h. Following are two examples of applying this heuristic to different vectors of deadlines for the problem in Example 3.3:

Heuristic(problem) = solution:

Kise (0, 1, 5, 6) = [$J_2 J_3 J_1 J_4$], two jobs tardy.
Kise (0, 8, 5, 4) = [$J_1 J_4 J_3 J_2$], one job tardy.

A genetic algorithm for CSRDD. In this work we developed a genetic algorithm based on the ideas presented in Davis, 1991, namely, steady-state reproduction without duplicates,

fitnesses measured by linear normalization, a uniform crossover operation, operation selection, and interpolated parameters.

Steady-state reproduction adds new individuals a few at a time, whereas the traditional method replaces the entire population with a new generation. Steady-state reproduction (attributed to Whitley, 1988, and Syswerda, 1989) is used to ensure that good individuals (and their good characteristics) survive. Steady-state reproduction without duplicates prevents children that are identical (in chromosome values) to a current member from joining the population.

Linear normalization is a fitness technique that creates fitness values by ordering the individuals in a population by their objective function evaluation. The assignment of fitnesses begins with a constant value and decreases the fitness linearly as it considers each individual in order. This technique prevents a super individual from dominating the population at the beginning of a run and yet differentiates between the various very good individuals that exist near the end of a run. Of course the values of the original constant and the decrement parameter influence the extent of these two phenomena.

Uniform crossover, an operator first described by Syswerda (1989), is a way to combine characteristics in ways that standard one- or two-point crossovers cannot. In a uniform crossover, two parents are selected and two children produced. Each bit position is considered independently and the parent that contributes the bit value for that position in the first child is determined randomly. The second child receives the value for that position from the other parent. While uniform crossover can destroy a good characteristic by mixing it with a bad string, it can also combine features that are widely dispersed across the string.

Since one-point crossover remains a good operator, however, we used both types of crossover in our genetic algorithm. Before creating a child, we randomly decide on which operator we wish to perform: uniform crossover, one-point crossover, or mutation. Each operator has an operator fitness and the probability of that operator being selected is proportional to that fitness. If one of the crossovers is selected, two parents are selected and two children are created.

If the mutation operator is selected, one parent is selected and a child created by forcing each bit to undergo a mutation with some small probability. The child or children created are checked against the current population for duplication, evaluated, and inserted into the population, replacing the worst members of the population. The new population is then reordered and new fitnesses created using the linear normalization technique.

As we did with the CFTS genetic algorithm, the initial population included one individual (the dummy, or seed) that was created form the actual problem data. The remaining individuals in the initial population were constructed by mutating the bits in the initial (dummy) chromosome. This initial mutation rate was set at 0.05 per bit.

We interpolate the following parameters over the course of the run: the decrement for linear normalization is increased, and the operator fitnesses are changed to favor crossovers and discourage mutations.

Table 3.10. Parameter values for genetic algorithm.

Population size: 100 Linear normalization decrement: 0.2 to 1.2 Probability of selecting mutation operator: 35% to 25% Probability of selecting uniform crossover operator: 40% to 30% Probability of selecting one-point crossover operator: 25% to 45% Mutation rate: 0.02 per bit

3.3.7 Empirical Tests and Results

In this section we describe the empirical tests conducted to test how well the heuristics and genetic algorithm perform.

Problem generation. We created problem sets using the due date assignment method of Hariri and Potts (1989). The release dates can also be created using a similar method. In this method, setup times and processing times of the jobs are determined first from random variables. Then, an estimate of the maximum completion time is made by simply summing the job

processing times. This makespan is used to define a specific ranges for the due dates and a range for the release dates. The due dates and release dates are sorted and the matching pairs are given to the jobs. By changing the parameters governing the definition of the ranges, problem sets with different characteristics can be created.

Different sets of 10 problems were created. The number of jobs per problem ranged from 15 to 100. The processing times ranged from 1 to 20 and the class setup times from 0 to 9. The jobs were randomly placed into a number of job classes, depending on problem size. The release dates and due dates were taken from a uniform distribution. The upper and lower bounds of this distribution were proportional to the sum of the job processing times. The proportions changed for each problem set. (See Table 3.11.)

Table 3.11. Data on problem sets.

Set	Problems	Jobs	Classes	Release date range	Due date range
KH301	10	30	4	0 - 0.4	0.4 - 0.6
KH302	10	30	4	0 - 0.4	0.6 - 1.0
KH151	10	15	4	0 - 0.4	0.4 - 0.6
KH501	10	50	5	0 - 0.4	0.4 - 0.6
KH303	10	30	4	0 - 0.4	0.2 - 1.0
KH304	10	30	4	0 - 0.6	0.2 - 1.0
KHMIXED2	10	30	4	0 - 0.4	0.4 - 0.6
KHMIXED3	10	30	4	0 - 1.0	0.4 - 1.2
KHMIXED4	10	30	8	0 - 1.0	0.4 - 1.2

Results. After numerical testing on these forty problems, it appears that the Kise and Kise extension heuristics and the R & M heuristics are fairly equal. A number of other heuristics were unable to find as many on-time jobs. Note that the lower bound was derived by using Kise's algorithm while ignoring all setup times.

Table 3.12. Average performance of heuristics.

Set	Jobs	Lower Bound	Kise	Extended Kise	RM	X-RM
KH301	30	8.3	12.0	10.7	11.6	11.6
KH302	30	1.1	4.9	2.7	4.2	4.3
KH151	15	4.8	6.8	6.4	6.6	6.7
KH501	50	12.6	19.5	16.0	17.9	17.9

Note: Performance is the average number of tardy jobs found by that heuristic on the ten problems in each problem set.

In addition to the problems where the release and due dates matched, we created a set of 30-job problems where no such correspondence existed. These ten problems had the same characteristics as the problems in the set KH301. On these problems the R & M heuristic performed slightly better than the extended Kise heuristic.

Table 3.13. Average performance of heuristics, non-matching release and due dates.

Set	Jobs	Kise	Extended Kise	RM	X-RM
KHMIXED2	30	7.7	5.3	4.5	4.7
KHMIXED3	30	11.1	10.2	8.9	8.7
KHMIXED4	30	10.0	9.2	7.2	7.0

Notes: Performance is the average number of tardy jobs found by that heuristic on the ten problems in each problem set.

Since the heuristics were finding good solutions, we decided to test the problem space genetic algorithm on 18- and 30-job problems that could not be solved well by the heuristics. These problems were instances of the problem described above in Example 3.6, where the extended Kise heuristic creates a schedule where a third of the jobs are tardy, although an optimal schedule has no tardy jobs. These results show that the problem space genetic algorithm is able to find good solutions. (In some of the following graphs we present our results with the number of on-time jobs, since the objective of the genetic algorithm was to maximize the number of on-time jobs.)

Table 3.14. Average performance of heuristics, hard problems.

Set	Jobs	Kise	Extended Kise	Genetic Algorithm[a] 1000	3000
Hard A	18	6.0	6.0	0.6	-
Hard B	30	10.0	10.0	4.5	1.5

Notes: Performance is the average number of TARDY jobs found by that heuristic on the five problems in each problem set.
a: Results reported at 1000 and 3000 individuals.

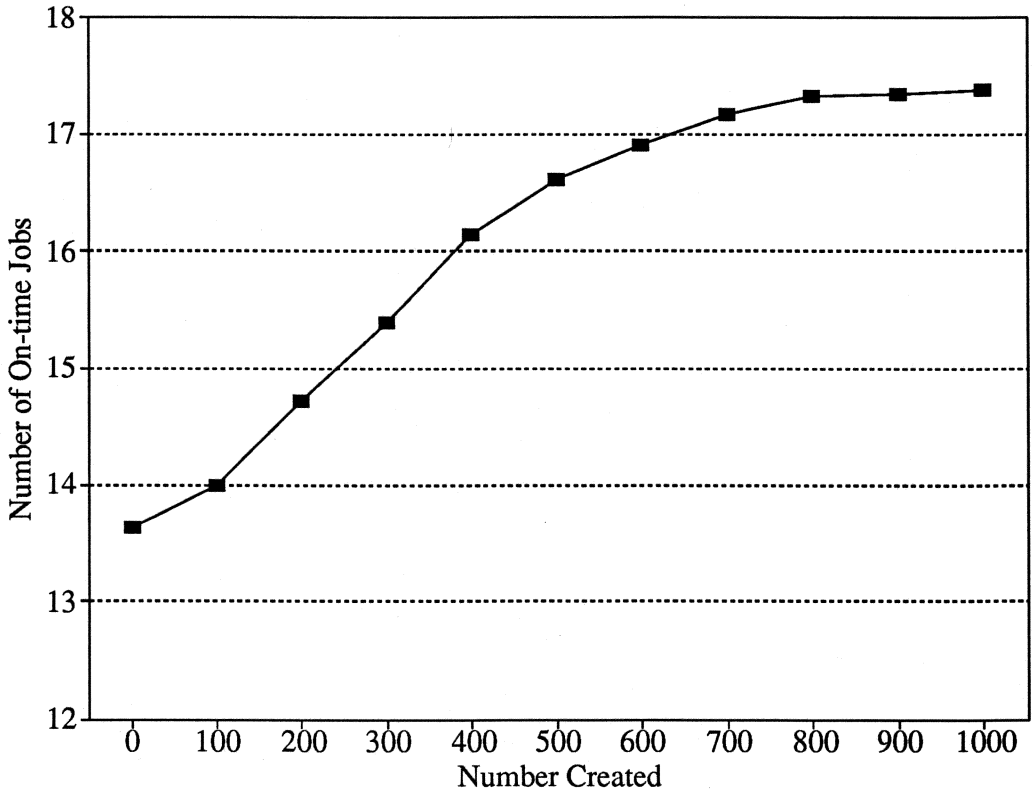

Figure 3.8. 18-job problems, Average number of ON-TIME jobs.

Table 3.15. 18-job problems, number of ON-TIME jobs, 10 runs of 1000:

Number of Individuals Created	Problem 1	2	3	4	5	Average
0	13.7	13.9	13.5	13.6	13.4	13.6
100	13.9	14.1	14.1	14.0	13.9	14.0
200	14.7	14.5	14.7	14.8	14.9	14.7
300	15.4	15.3	15.5	15.5	15.3	15.4
400	16.2	16.1	16.1	16.5	15.8	16.1
500	16.5	16.5	16.9	16.6	16.5	16.6
600	16.8	16.8	17.1	17.0	16.8	16.9
700	16.9	17.3	17.3	17.3	17.0	17.2
800	17.0	17.5	17.3	17.3	17.5	17.3
900	17.0	17.5	17.4	17.3	17.5	17.3
1000	17.1	17.5	17.5	17.3	17.5	17.4

Figure 3.9. 30-job problems, average number of ON-TIME jobs.

Table 3.16. 30-job problems, number of ON-TIME jobs; 3 runs of 3000

Number of Individuals Created	Problem 1	2	3	4	5	Average
0	20.3	20.3	20.7	21.3	20.7	20.67
100	21.3	21.0	21.0	21.7	21.3	21.27
200	21.7	21.7	21.0	22.0	21.7	21.60
300	22.0	22.0	21.7	22.0	22.0	21.93
400	22.7	22.3	22.3	22.7	22.3	22.47
500	23.3	23.0	22.7	23.0	23.3	23.07
600	24.0	24.0	23.0	23.3	23.7	23.60
700	24.0	24.0	24.0	23.7	23.7	23.87
800	25.0	24.3	24.7	24.7	24.7	24.67
900	25.0	24.7	25.0	24.7	25.0	24.87
1000	25.7	25.0	26.3	25.3	25.3	25.53
1100	26.0	26.0	26.7	26.3	25.7	26.13
1200	26.3	26.0	27.0	26.7	26.0	26.40
1300	26.7	26.7	27.7	26.7	26.3	26.80
1400	27.0	26.7	28.0	26.7	26.7	27.00
1500	27.3	27.0	28.0	27.0	26.7	27.20
1600	27.3	27.3	28.0	27.3	27.0	27.40
1700	27.7	27.3	28.3	27.3	27.0	27.53
1800	28.0	27.7	28.3	27.3	27.3	27.73
1900	28.0	27.7	28.3	28.0	27.3	27.87
2000	28.0	27.7	28.3	28.3	27.7	28.00
2100	28.3	27.7	28.3	28.3	27.7	28.07
2200	28.3	27.7	28.3	28.3	28.0	28.13
2300	28.3	27.7	28.3	28.3	28.3	28.20
2400	28.3	28.3	28.3	28.3	28.3	28.33
2500	28.3	28.3	28.3	28.7	28.3	28.40
2600	28.3	28.3	28.3	28.7	28.3	28.40
2700	28.3	28.3	28.7	28.7	28.3	28.47
2800	28.3	28.3	28.7	28.7	28.3	28.47
2900	28.3	28.3	28.7	28.7	28.3	28.47
3000	28.3	28.7	28.7	28.7	28.3	28.53

3.3.8 Extension to Minimizing Tardiness

In addition to simply minimzing the number of tardy jobs, it is often an objective of schedulers to minimize the total tardiness of the tardy jobs. To this end, we study the problem of

minimizing the total tardiness subject to a constraint on the number of tardy jobs, since the minimization of total tardiness usually leads to schedules where many jobs are tardy and are tardy by a small amount. Since finding the minimum number of tardy jobs is an NP-complete problem, we use our heuristic to set the value of the constraint. Then, within that limitation, we minimize the total tardiness.

We develop some further extensions of Kise's algorithm that create a set of tardy jobs and then insert the tardy jobs into the schedule of on-time jobs in order to reduce the total tardiness. We also use our genetic algorithm to search for schedules with low number of tardy jobs and low total tardiness.

The problem of minimizing tardiness subject to a minimal number of tardy jobs has been considered for the problem without class setups (or release dates) by Vairaktarakis and Lee (1993), who develop an algorithm to optimally schedule a given set of tardy jobs and an efficient branch-and-bound technique to find the optimal tardy set. Other researchers have studied dual criteria problems with the same primary objective. Emmons (1975) considered the problem of minimizing total flowtime subject to minimum number of tardy jobs, using a branch-and-bound algorithm to find optimal solutions. Shanthikumar (1983) examined the problem of minimizing the maximum lateness subject to minimum number of tardy jobs, also using a branch-and-bound algorithm.

Our problem, which includes both class scheduling and non-zero release dates, is an NP-complete problem. The tardiness heuristics that we use to find good solutions use the extended Kise heuristic to determine a set of tardy jobs. The heuristics also use the sequence of on-time jobs created by the extended Kise heuristic, pushing the jobs to the right, starting them as late as possible, and attempting to insert the tardy jobs into the gaps in this schedule.

The first heuristic (T_1) orders the tardy jobs by their release dates and attempts to interleave the two sequences of jobs, scheduling tardy jobs to start as soon as possible while maintaining the feasibility of each on-time job (whose completion time is constrained by the due date).

The second heuristic (T_2) considers every tardy job as a candidate for a gap between the partial schedule and the next on-time job, selecting the tardy job that yields the earliest start time for the next on-time job. Any remaining tardy jobs are scheduled by their release dates.

The third heuristic (T_3) was a modification of the second that scheduled the remaining tardy jobs using a version of the Minimum Waste heuristic (see Section 3.2) that didn't consider deadlines (since all jobs are tardy). This heuristic has been shown to perform well on flowtime criteria. We use this because minimizing the total tardiness of the set of tardy jobs is identical to minimizing the total flowtime of those jobs.

Due to the non-optimal nature of the extended Kise heuristic, it is possible that the tardiness heuristics will often be able to schedule a tardy job so that it finishes on-time (reducing its tardiness to zero). However, the primary objective of these heuristics is to reduce tardiness, not reduce the number of tardy jobs.

Results. We tested the heuristics on three problem sets, selected because the variance of the due dates meant that the schedules were more likely to have gaps in which to insert tardy jobs. Each solution technique was measured by the average total tardiness found by that heuristic on the ten problems in each problem set and the percent deviation of this average from the average tardiness found by the extended Kise heuristic. The performance of the genetic algorithm on each problem is the average of ten trials of one thousand new individuals.

Table 3.17. Data on new problem sets.

Set	Problems	Jobs	Classes	Release date range	Due date range
KH303	10	30	4	0 - 0.4	0.2 - 1.0
KH304	10	30	4	0 - 0.6	0.2 - 1.0

Table 3.18. Average performance of heuristics.

Set	Extended Kise	T_1	%	T_2	%	T_3	%
KH302	236.4	237.3	-0.38	234.7	0.72	231.5	2.07
KH303	758.3	756.6	0.22	718.4	5.26	704.1	7.15
KH304	920.0	869.0	5.54	859.4	6.59	844.0	8.26

Note: Performance is the average total tardiness and the percent improvement.

Table 3.19. Average performance of heuristics.

Set	Extended Kise	G.A.	%	G.A. 3000	%
KH302	236.4	229.0	3.13		
KH303	758.3	884.3	-16.62	695.3	8.30
KH304	920.0	771.9	16.10		

Note: Performance is the average total tardiness and the percent deviation.

3.3.9 Conclusions

In this section we have introduced a class scheduling problem that we call CSRDD. The problem is to minimize the number of tardy jobs where some jobs have non-zero release dates, and we assume that the release and due dates match. We have described a heuristic developed to find good solutions. We have discussed a problem space and a genetic algorithm to search this space. We have described our experimental results, from which we make the following conclusions:

Our extended Kise heuristic can find good solutions for instances of CSRDD. It can do this by considering the class setups in both the subalgorithm that decides on which job to make tardy when a new job is added and the option to insert the new job into the middle of the partial schedule in order to reduce the number of setups.

When the extended Kise heuristic cannot find good solutions, our problem space genetic algorithm can. By searching the problem space near the original problem, it can discover solutions that are improvements on the schedule constructed by Kise's algorithm.

Also in this section we discussed an extension of this problem to the problem of minimizing total tardiness in the presence of a constraint on the number of tardy jobs.

3.4 Flowtime with Setups and Release Dates

The third of the class scheduling problems that we consider has jobs with non-zero release dates, and the objective is to minimize the total flowtime. We develop some lower bounds and dominance properties and examine some heuristics for finding good solutions to the problem. We discuss a problem space genetic algorithm that can improve the performance of a look-behind dispatching rule. For this problem we also developed a search technique for comparison purposes.

3.4.1 Introduction

This problem, like the others we have examined, is motivated by considering the scheduling of a semiconductor test area. We have class setups, arriving jobs, and an objective that mirrors the goal of management to minimize work-in-process inventory.

We will examine the problem of minimizing total flowtime when the jobs have non-zero release dates. This strongly NP-complete problem is a look-behind scheduling model, where we are interested in scheduling a machine by considering the jobs that will be arriving at the machine soon.

In addition to a look-behind scheduling rule, we will consider the use of a problem space genetic algorithm (similar to those developed for CFTS and CSRDD) that can improve the performance of this rule by adjusting the parameters of the rule. Due to the structure of the FTSRD problem, we will also use a decomposition heuristic as a means of comparing solution

quality. The decomposition heuristic is a search technique that considers a sequence of subproblems at each move.

In the next section we will introduce the notation and problem formulation. After that we will mention some of the previous research on the scheduling problem under consideration (a review of the literature on class scheduling and genetic algorithms can be found in Chapter 2), examine some lower bounds and dominance properties that can be used in a branch-and-bound technique, discuss our heuristics (including sequencing rules and the genetic algorithm), and report on the experimental results.

3.4.2 Notation and Problem Formulation

We use the same notation as that for the CFTS and CSRDD problems (Sections 3.2.3 and 3.3.3), except that the jobs do not have due dates. The FTSRD problem is to find a sequence that minimizes $\sum C_j$ subject to the constraint that $C_j \geq r_j + p_j$. We will see that FTSRD is NP-complete and has not been previously considered in the literature on class scheduling.

We make two assumptions in the analysis of the problem. One, a class setup for a job can begin before the job is available. Two, although all of the release dates are known, the processing for a job cannot begin until the release date. These conditions are motivated by our consideration of the one-machine problem as part of the job shop scheduling problem.

3.4.3 Background

The one-machine problem of minimizing total flowtime when the jobs have non-zero release dates has been previously studied in the case where no sequence-dependent setups are present. The problem is simple if the jobs are preemptive, that is, if a job that has begun processing can be interrupted by another job and then resumed later. In this case, the optimal policy at the next decision point is to schedule the job with the shortest remaining time. The set of decision points includes all job release times and completion times. If the jobs are non-

We note here that matching processing times and release dates do not imply a dominance property. Even if $p_j < p_i$ implies $r_j \leq r_i$ (or even if the release dates are identical) within each class, we do not have an optimal order for the jobs in each class. Consider the following instance:

	J_j	p_j	r_j
G_1	J_1	3	0
	J_2	4	0
	J_3	4	0
G_2	J_4	1	6
	J_5	1	6
	J_6	1	6

$s_{01} = s_{21} = s_{02} = s_{12} = 1.$

The optimal solution is $[J_2, J_4, J_5, J_6, J_1, J_3]$, with a total flowtime of 59. The best solution in which G_1 is in SPT order are $[J_1, J_4, J_5, J_6, J_2, J_3]$, with a total flowtime of 60 (see Figure 3.10).

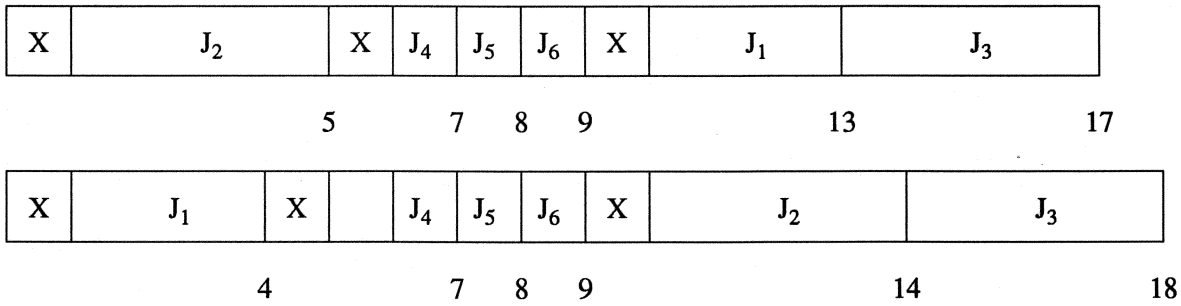

Figure 3.10. Optimal schedule and best SPT schedule.

A number of dominance properties have been suggested for the $1 / r_j / \Sigma C_j$. We have modified the properties of Dessouky and Deogun (1981) for use in our search. We make the assumption here that the class setups satisfy the triangle inequality: $s_{ac} \leq s_{ab} + s_{bc}$ for all G_a, G_b, G_c.

Suppose that we are at a node in the search tree with a partial schedule σ that ends at time t with a job in class G_c and a set K of unscheduled jobs. If J_j is in K and in G_b, define the following *earliest start time*:

$$t_j = \max \{t + s_{cb}, r_j\}.$$

Note that if $t + s_{cb} \leq r_j$, the necessary class setup can be completed before the job arrives. At this node we can use any of the following dominance properties:

Property 3.1. Let J_i be the shortest job in K. If J_j is in G_b and J_i is in G_a, the node (σ, J_j) is dominated by (σ, J_i) if $t_i + s_{ab} \leq t_j$.

Property 3.2. The node (σ, J_j) is dominated by (σ, J_i) if $t_i + p_i + s_{ab} \leq r_j$.

Property 3.3. If J_j is in G_b and J_i is in G_a, the node (σ, J_j) is dominated by (σ, J_i) if all of the following statements are true:

(a) $t_i + p_i \leq t_j + p_j$,

(b) $t_i + p_i + s_{ae} \leq t_j + p_j + s_{be}$ for all job classes $G_e : G_e \cap K \neq \{\}$,

(c) $s_{eb} + p_j \leq s_{ea} + p_i$ for all $G_e : G_e \cap K \neq \{\}$, and

(d) $s_{eb} + p_j + s_{bd} \leq s_{ea} + p_i + s_{ad}$ for all $G_e : G_e \cap K \neq \{\}$, and $G_d : G_d \cap K \neq \{\}$.

Proofs: Property 3.1. Take any schedule $(\sigma, J_j, \sigma_1, J_i, \sigma_2)$, where there are m jobs in σ_1. If we move J_i before J_j, the new completion time of J_i is no greater than the old completion time of J_j; thus the flowtime of J_i is decreased by at least $(m + 1)p_i$ (since J_i is the shortest job). Meanwhile, we delay only the start of J_j and the m jobs in σ_1 by at most p_i (since $t_i + p_i + s_{ab} - t_j \leq p_i$). The jobs in σ_2 are not delayed at all (the triangle inequality of the setups insures this).

Property 3.2. Take any schedule $(\sigma, J_j, \sigma_1, J_i, \sigma_2)$. We can move J_i before J_j without delaying J_j, decreasing the flowtime of J_i and possibly that of the jobs in σ_2.

Property 3.3. Take any schedule $(\sigma, J_j, \sigma_1, J_i, \sigma_2)$. Note that the earliest start times of J_i and J_j are before σ_1. Interchange J_i and J_j. The new completion time for J_i is not greater the old completion time of J_j (by condition a). By condition b (if the first job of σ_1 is in G_e), the jobs in σ_1 are not delayed. Then, by condition c (if the last job of σ_1 is in G_e), the new completion time

for J_j is not greater than the old completion time of J_i. Finally, the last condition (if the first job of σ_2 is in G_d) implies that no jobs in σ_2 are delayed.

In any of these cases, we can take a schedule that starts with (σ, J_j) and find a schedule which starts with (σ, J_i) and which has less total flowtime. Thus, it is clear that the first node is dominated and that we do not need to search that part of the tree.

Lower bound. A branch-and-bound procedure needs a lower bound. A good lower bound is important to efficiently finding solutions. We develop two bounds: the first concentrates on the release dates, the second on the setup times.

The first lower bound for a node completely ignores the class setups. Instead, it solves the associated $1 / r_j$, preemption $/ \sum C_j$ problem with the SRPT rule and adds the optimal flowtime to that of the partial schedule in the node.

The second lower bound separates the setup times from the job processing times. We assume that each class will have exactly one remaining setup. If we further assume that this setup will the shortest possible, then we can easily sequence the job classes to minimize the contribution of the setup times to the total flowtime. The unscheduled jobs are scheduled by SPT without regard to their release dates. The two sums are added to the flowtime of the partial schedule in the node.

The first lower bound should work well when the interarrival times are large, and the second bound should be useful at nodes where all of the unscheduled jobs are available.

Testing. For 30-job problems, the branch-and-bound procedure required excessive computation time to find an optimal. Still, improved lower bounds could be found by truncating the branch-and-bound search in the following way: We prevent the search from moving below a certain depth in the tree and take the lower bound at this depth as the objective function value. If we continue to do this, the truncated search returns the lowest lower bound at this depth. This will be a lower bound on the optimal solution and will be better than the lower bound computed at the root node.

Dispatching rules. The SPT rule is known to minimize the total flowtime for $1 // \Sigma C_j$ (Smith, 1956). Since we are studying a total flowtime problem, we are most interested in SPT-like heuristics. We will develop a rule that considers the waste associated with a job (the *waste* is the sum of the idle time and setup time incurred if the job is scheduled next). We propose the look-behind dispatching rule Shortest Waste, "Among the jobs with the minimum waste, schedule the shortest one." (This rule is similar to the Minimum Waste algorithm for CFTS.)

Let us define a few relevant variables: t is the current time, the completion of the last scheduled job; G_c is the class of the last scheduled job, and the *waste* of an unscheduled job J_j in G_b is

$$w_j = \max \{r_j - t, s_{cb}\}.$$

Our dispatching rule can be now stated:

Shortest Waste:

Among all unscheduled J_j, select the job with the minimum w_j. Break any ties by selecting the one with the minimum p_j.

Decomposition. For the FTSRD problem, we decided to implement a search heuristic for comparison purposes. A decomposition heuristic for finding good solutions to sequencing problems was introduced by Chambers et al. (1992). The heuristic is a type of local search. It begins with an initial sequence, and forms new sequences until it finds a local minimum. The critical step is the decomposition of the problem into subproblems that depend upon the current solution. A new solution is generated by combining the optimal or near-optimal solutions to each subproblem. The heuristic thus makes very good moves through the search space; only a few moves are needed before convergence is reached.

Consider, for example, a 12-job problem. Start with some initial solution to the problem. Select the first six jobs of this solution and find a good solution to the 6-job subproblem. Take the first three jobs (in order) of this subproblem solution as the first three jobs of the new solution. Then combine the remaining three jobs from the subproblem and with the next three

jobs from the initial solution. This forms a new 6-job subproblem. Continue solving subproblems and building the new solution until all of the jobs have been considered.

We use this technique in order to find good solutions against which we could measure the performance of our other heuristics. (We did not feel that this type of search would be as effective on deadline-oriented CFTS and CSRDD problems.) We use a branch-and-bound technique with an approximate dominance property to generate near-optimal solutions to the subproblems. The algorithm has two parameters. We need to select m as the size of the subproblems which we will solve; the larger the value, the better our subproblem solutions will be (at the expense of computation time). We also select f as the number of jobs from the subproblem solution that will be fixed into the new solution. A smaller f requires that more subproblems be solved per step. The following steps outline the procedure.

Step 1. Set m and f. (We want f to divide $n - m$.) Let σ be the ERD schedule.
Step 2. Take the first m jobs of σ. Let π be an empty schedule.
Step 3. Solve the m-job subproblem by branch-and-bound.
Step 4. Append the first f jobs of the solution to π.
Step 5. If there are any unconsidered jobs in σ, take the next f jobs from σ, add to the $m - f$ remaining from the subproblem, and go to Step 3; else go to Step 6.
Step 6. Append the $m - f$ remaining jobs of the solution to π. If π is a better schedule than σ, let $\sigma = \pi$ and return to Step 2; else go to Step 7.
Step 7. The solution found by the heuristic is σ.

We experimented with different values of (m, f). The best results (in terms of computation time and solution quality) were achieved with (9, 3) and (15, 5). We used our branch-and-bound algorithm with only the second lower bound and only the following approximate dominance property:

Property 3.4. Given a partial schedule σ, (σ, J_i) dominates (σ, J_j) if J_i and J_j are in the same class, J_i is the shortest unscheduled job in that class, and $w_i \leq w_j$.

This property is a simple extension of a previously considered dominance rule (see, for instance, Dessouky and Deogun, 1981). Unlike the dominance rules that we use in the full

branch-and-bound procedure, it has the advantage of being quick to check, since there are fewer unscheduled jobs from the same class.

In the section on computational results, we will discuss how well the decomposition heuristic performs.

A problem space genetic algorithm. In this problem, we consider the problem space defined over the problem release dates. A point in this space is an n-element vector of non-negative real numbers. When a heuristic is applied to an instance of FTSRD, it uses the actual release dates to generate a solution. If, however, we adjust the release dates of the problem, we can change the sequence created by the heuristic. This sequence can be evaluated as a schedule by using the actual problem data. We can associate, therefore, with the vector of adjusted release dates a performance value: the total flowtime of the schedule that was created. Moreover, we can search the space of adjusted release dates to find good schedules. This exploration is the objective of the problem space genetic algorithm. Our purpose is to show that the performance of a simple heuristic can be improved with a smart-and-lucky search like a genetic algorithm.

We will use the Shortest Waste heuristic to convert a vector of adjusted release dates into a sequence of jobs. The optimal solution is within the range of this heuristic: if each adjusted release date equals the actual start time of the job in an optimal solution, the Shortest Waste heuristic will schedule the jobs in the optimal order, since at any time, the job with the shortest waste will be the one with the next adjusted release date, which is the job with the next optimal start time.

As we did for CFTS and CSRDD, we will use a steady-state genetic algorithm. The initial population is formed by mutating a source individual that is the digital representation of the actual release dates. After empirical testing on a number of problem instances, we decided on the following parameters: The population size is 100 individuals. The four operators are uniform crossover, one-point crossover, small mutation, and large mutation; all have the same probability of being selected. In the small mutation, a bit is flipped with 2% probability; in the large, the probability increases to 50%. The algorithm uses tournament selection to identify parents. See

Davis (1991) or Goldberg (1989) for more information about these aspects of the genetic algorithm.

Example 3.7. The following problem is used to illustrate some of the issues we have discussed so far.

j	r_j	p_j	i
1	0	5	1
2	0	4	2
3	5	5	2
4	6	2	1

$s_{01} = s_{02} = 1.\ s_{12} = 1.\ s_{21} = 4.$

The ERD sequence is $[J_1\, J_2\, J_3\, J_4]$, with a total flowtime of 55. The EFT sequence is $[J_2\, J_3\, J_4\, J_1]$, with a total flowtime of 52. The Shortest Waste sequence is identical in this case. If we adjust the release dates to (1, 2, 5, 6), the Shortest Waste sequence is $[J_1\, J_4\, J_2\, J_3]$, with a flowtime of 45.

In the branch-and-bound algorithm, we compute lower bounds at the root node. The first lower bound for the entire problem is the SRPT schedule, total flowtime of 39, shown below (J_1 is preempted at time 6 by J_4):

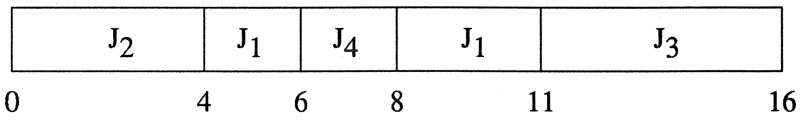

Figure 3.11. The SRPT schedule.

The second lower bound (Longest Weighted Batch Size plus SPT) is computed as follows:

Batch sizes: $b_1 = 2.\ b_2 = 2.$
Shortest setups: $s_1 = s_{01} = 1.\ s_2 = s_{02} = 1.$
$\sum s_i B_i = 1(4) + 1(2) = 6.$
Processing times in SPT order: 2, 4, 5, 5.
Completion times: 2, 6, 11, 16. $\sum C_j = 35.$
Lower bound = 35 + 6 = 41.

3.4.5 Empirical Testing

Problem generation. In order to test the heuristics, four sets of ten problems were created. The characteristics of the sets are shown below (processing, interarrival, and setup times randomly selected from uniform distributions with the given ranges):

Table 3.20. Data on problem sets

Set	Problems	Jobs	Classes	Processing Times	Interarrival Times	Setup Times
FT151	10	15	5	1,20	1,10	0,9
FT301	10	30	5	1,20	1,15	0,9
FT302	10	30	10	1,20	1,15	5,9
FT304	10	30	10	1,15	1,20	5,9

Results. In this section we will discuss how well our solution techniques performed. (See Table 3.21.) The branch-and-bound could find optimal solutions on only the 15-job problems. On the 30-job problems, we used the decomposition heuristic with parameters (9, 3) to quickly generate solutions and measured the performance of other heuristics against these solutions. The (15, 5) decomposition was much slower than the (9, 3) decomposition, since the time necessary to solve each subproblem grew exponentially. Still, it found slightly better solutions, and the processing time was reasonable (although it varied from problem to problem) if the heuristic dominance property and second lower bound were used.

Table 3.21. Performance of heuristics.

Problem Set	Shortest Waste	(9,3)	(15,5)	Genetic Algorithm
FT151	1.095	1.005	-	1.010
FT301	1.067	1.000	0.992	1.006
FT302	1.066	1.000	0.995	1.005
FT304	1.031	1.000	0.994	1.005

Notes: Performance measured against optimal solution for FT151. Against decomposition (9,3) for 30-job problems. All performances are average ratios over 10 problems. Performance of genetic algorithms averaged over three runs of 3000 individuals.

Although the initial lower bounds for the 30-job problems were not good, we were able to improve them using the branch-and-bound tree to show that the decomposition (9, 3) heuristic was within ten percent of the optimal flowtime.

The Shortest Waste heuristic found good solutions very quickly and generally performed better than other dispatching rules. On the 15-job problems, the genetic algorithm was not an effective heuristic, since it required more computation time than the branch-and-bound and could not always find optimal solutions.

On the 30-job problems, the problem space genetic algorithm found solutions better than Shortest Waste and as good as the decomposition heuristic. The computation time was slightly longer for a 3000-individual search than for a (15, 5) decomposition, but a 1000-individual search was much shorter and found solutions with little increase in total flowtime. The exponential nature of genetic search is exhibited in Figure 3.12. (In other testing, we found that the genetic algorithm was not as effective when using a simple Earliest Release Date rule to create sequences).

All programs were run on a 386 PC. Decreases in times were achieved when the programs were run on a 486 PC, and further decreases could be achieved on a more powerful machine. Except for the 30-job branch-and-bound (which we could not solve), we do not consider processing times to be a significant obstacle. The numbers in Table 3.22 are offered only for comparison purposes.

Table 3.22. Typical computation times

FT302.1
Decomposition (9,3): 17.8 seconds
Decomposition (15,5): 338.67 seconds
1000-individual GA: 134.62 seconds
3000-individual GA: 350.37 seconds
Shortest Waste: < 0.1 seconds

FT151.1
Branch-and-bound: 10.71 seconds
Decomposition (9,3): 0.8 seconds
3000-individual GA: 149.24 seconds
Shortest Waste: < 0.1 seconds

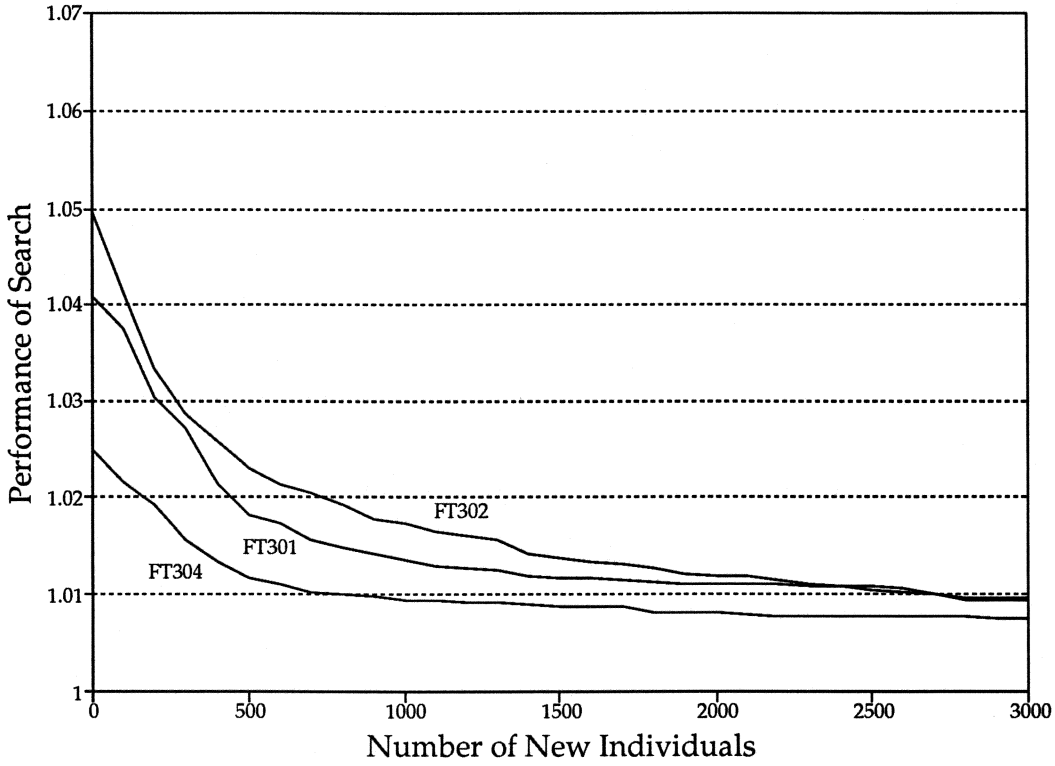

Figure 3.12. Performance of Shortest Waste Genetic Algorithm
Performance measured against decomposition (9,3).

3.4.6 Special Case

In this section we consider a particular special case of the problem which may be useful in certain manufacturing situations. We will assume that there exist exactly two job classes and that the job processing times are equal within each class.

Specifically, we study the following instance: $p_j = p$ for all J_j in G_1, $p_j = q$ for all J_j in G_2. Since all of the jobs in a class have identical processing times, we may order them by ERD. We note here that all of the release times are integer.

We will describe a pseudo-polynomial dynamic program to solve the problem, a special case of the strongly NP-complete FTSRD problem.

Dynamic programming. We can use a dynamic program for two reasons: there are only two classes, and we have an ordering for the jobs in each class. According to Monma and Potts (1989), this *ordered batch scheduling problem* can be solved in pseudo-polynomial time. The following dynamic program interleaves the classes. The state variable in the dynamic program corresponds to a partial schedule that consists of the first i_1 jobs from G_1 and the first i_2 jobs from G_2 and that ends before or at a specific time with a job from a specific class (if it is cheaper to end sooner, that schedule should take precedence). At each point in the state space, we will measure the total flowtime of the scheduled jobs. The recursion determines the best partial schedule to which we should add the specified job.

Algorithm 3.3. Let $f(t, a, i_1, i_2)$ be the minimum flowtime of a partial schedule where the last jobs ends at or before time t, there are i_1 jobs from G_1 and i_2 jobs from G_2, and the last scheduled job is from G_a. $t = 0, \ldots, R$. $a = 1, 2$. $i_1 = 0, \ldots, n_1$, $i_2 = 0, \ldots, n_2$. Renumber the jobs so that $j \leq n_1$ if J_j is in G_1, $r_1 \leq \ldots \leq r_{n1}$, and $j > n_1$ if J_j is in G_2, $r_{n1+1} \leq \ldots \leq r_n$. R is some upper bound on the makespan of a schedule. We can find one such R by scheduling all jobs in ERD order, performing a class setup in front of every job. We also have an upper bound: $R \leq \max\{r_j\} + \sum p_j + n_1 s_{21} + n_2 s_{12}$.

Initialization:

$f(t, a, i_1, i_2) = \infty$ if $t < 0$.

$f(t, 1, 0, i_2) = \infty$ for all t and for $i_2 > 0$.

$f(t, 2, i_1, 0) = \infty$ for all t and for $i_1 > 0$.

$f(t, 1, 1, 0) = \infty$ for $t < \max\{s_{01}, r_j\} + p$, if J_j is the first job in G_1 ($j = 1$).

$f(t, 1, 1, 0) = \max\{s_{01}, r_j\} + p$ for $t \geq \max\{s_{01}, r_j\} + p$, if J_j is the first job in G_1 ($j = 1$).

$f(t, 2, 0, 1) = \infty$ for $t < \max\{s_{02}, r_j\} + q$, if J_j is the first job in G_2 ($j = n_1 + 1$).

$f(t, 2, 0, 1) = \max\{s_{02}, r_j\} + q$ for $t \geq \max\{s_{02}, r_j\} + q$, if J_j is the first job in G_2 ($j = n_1 + 1$).

Iteration: ($i_1 + i_2 > 1$)

$f(t, 1, i_1, i_2) = \min\{f(t\text{-}p, 1, i_1\text{-}1, i_2) + t, f(t\text{-}p\text{-}s_{21}, 2, i_1\text{-}1, i_2) + t, f(t\text{-}1, 1, i_1, i_2)\}$ if $t \geq r_j + p$, where $j = i_1$.

$f(t, 1, i_1, i_2) = \infty$ if $t < r_j + p$, where $j = i_1$.

$f(t, 2, i_1, i_2) = \min \{f(t\text{-}q, 2, i_1, i_2\text{-}1) + t, f(t\text{-}q\text{-}s_{12}, 1, i_1, i_2\text{-}1) + t, f(t\text{-}1, 2, i_1, i_2)\}$ if $t \geq r_j + q$, where $j = n_1 + i_2$.

$f(t, 2, i_1, i_2) = \infty$ if $t < r_j + q$, where $j = n_1 + i_2$.

Answer: the optimal total flowtime is $\min \{f(R, 1, n_1, n_2), f(R, 2, n_1, n_2)\}$.

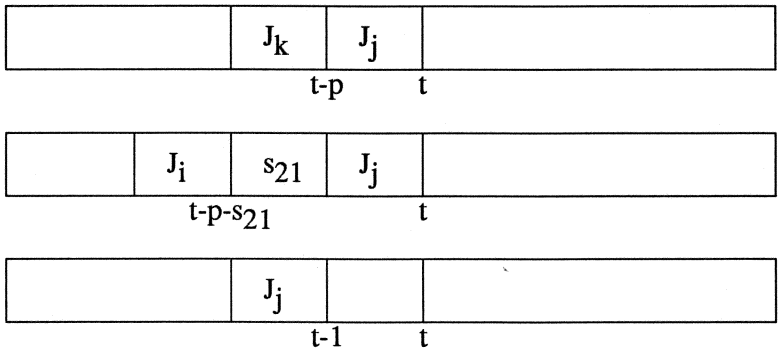

Figure 3.13. Schedules for iteration (J_j and J_k in G_1 and J_i in G_2).

In the iteration, the first term being considered for $f(t, 1, i_1, i_2)$ is the total flowtime of the partial schedule formed by adding the J_j ($j = i_1$) to a schedule ending at or before $t\text{-}p$ with a job in G_1. The second term is the flowtime if J_j is added to a schedule ending at or before $t\text{-}p\text{-}s_{21}$ with a job in G_2. The third term corresponds to a schedule that ends at or before t-1 with J_j. If $t < r_j + p$, there is no feasible schedule that ends with J_j; thus, $f(t, 1, i_1, i_2)$ is set to infinity. The iteration is similar for the $f(t, 2, i_1, i_2)$.

The effort for each point in the state space is constant. The effort for the entire program is thus proportional to the number of points in the state space. This is $O(Rn^2)$. Since R is bounded by a polynomial function of the problem data, the algorithm is pseudo-polynomial. The dynamic program can be implemented to find the optimal objective function value with memory requirements that are $O(Rn)$: If we are determining the values for points with $i_1 + i_2 = k$, any points in the space where $i_1 + i_2 < k - 1$ are ignored. If we define $f_k(t, a, i_1) = f(t, a, i_1, k - i_1)$, we need to keep only f_k and $f_{k\text{-}1}$ at any time.

Test problems. We generated 70 test problems in order to test the dynamic program on a range of problem sizes. The data for the problem sets are summarized in Table 3.23 (10 problems in each set).

Table 3.23. Data about problem sets for special case.

Problem Set	Number of jobs	p	q	Class setup	Interarrival times
FT201	20	1	1	1	1-3
FT307	30	1	1	1	1-3
FT308	30	2	3	1	2-5
FT309	30	4	2	3	4-8
FT401	40	1	1	1	1-3
FT507	50	1	1	1	1-3
FT601	60	1	1	1	1-3

Results of special case dynamic program. The dynamic program finds optimal solutions in time that is nearly proportional to Rn^2 (see Table 3.24).

One drawback of the dynamic program is the amount of memory required to perform the algorithm; we were able to solve only 60-job problems. In addition, it requires significant processing time on a 386 personal computer. These problems can be overcome, however, since a larger computer could handle more memory (the amount required is $O(Rn)$), and would run more quickly.

Table 3.24. Results of dynamic program

Problem Set	Number of jobs	Average R	Average computation time
FT201	20	41.1	0.808
FT307	30	61.4	2.270
FT308	30	108.1	3.487
FT309	30	185.5	5.899
FT401	40	81.8	4.698
FT507	50	101.6	9.079
FT601	60	121.4	14.836

Extensions of special case. The dynamic program can be modified to solve any FTSRD problem where there exists a natural order for the jobs within each class. This includes problems where the jobs in each class have the same processing time (as we have discussed) or problems where all of the jobs have the same release date (order the jobs in each class by SPT). In any of these cases we have an ordered batch scheduling problem. If each class has a natural order, the dynamic program can be used to interleave these sequences since we have an ordered batch scheduling problem.

Recall from the example presented in Section 3.4.4 that matching processing times and release dates do not give us a natural order for a class.

3.4.7 Conclusions

In this research we have studied a computationally difficult class scheduling problem. The objective is to minimize the total flowtime of a set of jobs that have non-zero release dates. We examined a number of techniques to solve the problem, including a branch-and-bound search, look-behind dispatching rules, a decomposition heuristic, and a problem space genetic algorithm. We were interested in determining how this type of genetic algorithm can be used to find good solutions for another class scheduling problem.

Our results are as follows: While we did develop lower bounds and a number of dominance properties, our branch-and-bound approach was unable to solve any 30-job problems. The decomposition heuristic was a successful technique, locating solutions of high quality. The Shorest Waste heuristic could sometimes generate good solutions. However, by incorporating these rules in a genetic algorithm that searched the space of adjusted release dates, we could find much better solutions.

From these results we can conclude two things: For the one-machine class scheduling problem we call FTSRD both the problem space genetic algorithm and the decomposition heuristic can find good solutions in reasonable time. Additionally, look-behind rules may be

useful for job shop scheduling, especially on a bottleneck machine which undergoes class setups and where jobs continue to arrive while the machine is processing.

3.5 Chapter Summary

In this chapter we have presented the results of research into three one-machine class scheduling problems. We can make a number of observations about this research. 1. None of these three problems have been previously considered in the literature. 2. We have presented analytical results and developed extended heuristics for each of these problems. 3. All three problems are motivated by the semiconductor test area job shop environment, and the extended heuristics developed for these problems may be useful as dispatching rules in the general job shop scheduling problem. 4. Our problem space genetic algorithm is a robust approach, able to find good solutions over a variety of one-machine class scheduling problems, and should be applicable to other difficult combinatorial and scheduling problems.